REMARKS/ARGUMENTS

In the Office Action mailed April 21, 2005, claims 1-23 were rejected under 35 U.S.C. 103(a) as being unpatentable over Cooper et al. (U.S. Patent No. 6,829,713) in view of Fruehling et al. (U.S. Patent No. 6,625,688). No claims have been added, canceled, or amended. Claims 1-23 remain pending.

I. Claim Rejections under 35 U.S.C. § 103(a)

Claims 1-23 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Cooper et al (US Patent No 6,829,713 B2) in view of Fruehling et al. (US Patent No 6,625,688 B1).

Applicant respectfully traverses the Examiner's rejections under 35 U.S.C. § 103(a), on the grounds that Cooper et al is not prior art under 35 U.S.C. § 102(e). The effective date of Cooper et al is December 30, 2000. However, as detailed in the attached 37 CFR 1.131 declaration (Exhibit A) and its attachments (Exhibits B and C), the present invention was reduced to practice prior to this date. Consequently, Cooper et al is not prior art under 35 U.S.C. § 102(e), and therefore, cannot be combined with other art under 35 U.S.C. § 103(a) to reject the claims. For this reason, applicant respectfully requests reconsideration of the 35 U.S.C. § 103(a) rejections. Withdrawal of these rejections is respectfully requested.

In view of the foregoing amendments and remarks, all pending claims are believed to be allowable and the application is in condition for allowance. Therefore, a Notice of Allowance is respectfully requested. Should the Examiner have any further issues regarding this application,

the Examiner is requested to contact the undersigned attorney for the applicant at the telephone number provided below.

Respectfully submitted,

MERCHANT & GOULD P.C.

Lawrence E. Lycke
Registration No. 38,540
Direct Dial: 206.342.6215

MERCHANT & GOULD P.C.
P. O. Box 2903
Minneapolis, Minnesota 55402-0903
206.342.6200

EXHIBIT A

S/N 09/882,076 PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants	Stephane G. Plante et al.	Examiner:	Suresh Suryawanshi
Application No.:	09/882,076	Group Art Unit:	2115
Filed:	June 15, 2001	Docket No.:	50037.08US01
Title:	METHOD AND SYSTEM FOR USING IDLE THREADS TO ADAPTIVELY THROTTLE A COMPUTER		

EXHIBIT A

DECLARATION UNDER 37 CFR §1.131

We, Stephane G. Plante, John D. Vert, and Jacob Oshins, declare as follows:

1. We are joint inventors named on U.S. Patent Application Serial No. 09/882,076 filed June 15, 2001 (hereinafter, "this application").

2. I am aware that a Final Office Action was mailed in this application on April 21, 2005 and that, in this Final Office Action, all pending claims were rejected either as being obvious under 35 USC § 103(a) in view of U.S. Patent No. 6,829,713 B2 Cooper et al. (hereinafter, "Cooper") and further in view of U.S. Patent No. 6,625,688 B1 Fruehling et al. (hereinafter, "Frueling").

3. I am aware of an Amendment in the present application being filed in response to this subsequent Office Action and that this declaration is attached to that Amendment as Exhibit A.

4. The invention set forth in all claims submitted in this application, whether pending, original or previously amended, was conceived and actually reduced to practice by us in this country at least prior to December 30, 2000. Exhibit B, attached hereto, is a document that lists versions dating from September 3, 2000 to September 25, 2000 and describes adaptive throttling. The document includes a general description of adaptive throttling as well as including computer-executable algorithms (e.g., see algorithm on pages 8-9) that illustrate the constructive reduction to practice of the invention. This document therefore contains a dated description that evidences conception and reduction to practice of the claimed invention at least as early as September 3, 2000.

5. As further evidence of conception and reduction to practice Exhibit C, attached hereto, illustrates a redlined updated version of the document in exhibit B we made on October 29, 2000. As evidenced by the actual working computer-executable algorithms provided, this document further illustrates the conception and reduction to practice of the invention at least prior December 30, 2000.

6. We hereby declare that all statements made herein of our own knowledge are true and that all statements made on information and belief are believed to be true; and further that statements are made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such false statements may jeopardize the validity of the application or any patent issued thereon.

Date July 20th, 2005

Stephane G. Plante

Date _____

John D. Vert

Date _____

Jacob Oshins

S/N 09/882,076 **PATENT**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants	Stephane G. Plante et al.	Examiner:	Suresh Suryawanshi
Application No.:	09/882,076	Group Art Unit:	2115
Filed:	June 15, 2001	Docket No.:	50037.08US01
Title:	METHOD AND SYSTEM FOR USING IDLE THREADS TO ADAPTIVELY THROTTLE A COMPUTER		

EXHIBIT A

DECLARATION UNDER 37 CFR §1.131

We, Stephane G. Plante, John D. Vert, and Jacob Oshins, declare as follows:

1. We are joint inventors named on U.S. Patent Application Serial No. 09/882,076 filed June 15, 2001 (hereinafter, "this application").

2. I am aware that a Final Office Action was mailed in this application on April 21, 2005 and that, in this Final Office Action, all pending claims were rejected either as being obvious under 35 USC § 103(a) in view of U.S. Patent No. 6,829,713 B2 Cooper et al. (hereinafter, "Cooper") and further in view of U.S. Patent No. 6,625,688 B1 Fruehling et al. (hereinafter, "Frueling").

3. I am aware of an Amendment in the present application being filed in response to this subsequent Office Action and that this declaration is attached to that Amendment as Exhibit A.

4. The invention set forth in all claims submitted in this application, whether pending, original or previously amended, was conceived and actually reduced to practice by us in this country at least prior to December 30, 2000. Exhibit B, attached hereto, is a document that lists versions dating from September 3, 2000 to September 25, 2000 and describes adaptive throttling. The document includes a general description of adaptive throttling as well as including computer-executable algorithms (e.g., see algorithm on pages 8-9) that illustrate the constructive reduction to practice of the invention. This document therefore contains a dated description that evidences conception and reduction to practice of the claimed invention at least as early as September 3, 2000.

5. As further evidence of conception and reduction to practice Exhibit C, attached hereto, illustrates a redlined updated version of the document in exhibit B we made on October 29, 2000. As evidenced by the actual working computer-executable algorithms provided, this document further illustrates the conception and reduction to practice of the invention at least prior December 30, 2000.

6. We hereby declare that all statements made herein of our own knowledge are true and that all statements made on information and belief are believed to be true; and further that statements are made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such false statements may jeopardize the validity of the application or any patent issued thereon.

Date_____ _____
Stephane G. Plante

Date 7/20/05 _____
John D. Vert

Date_____ _____
Jacob Oshins

S/N 09/882,076 PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants	Stephane G. Plante et al.	Examiner:	Suresh Suryawanshi
Application No.:	09/882,076	Group Art Unit:	2115
Filed:	June 15, 2001	Docket No.:	50037.08US01
Title:	METHOD AND SYSTEM FOR USING IDLE THREADS TO ADAPTIVELY THROTTLE A COMPUTER		

EXHIBIT A

DECLARATION UNDER 37 CFR §1.131

We, Stephane G. Plante, John D. Vert, and Jacob Oshins, declare as follows:

1. We are joint inventors named on U.S. Patent Application Serial No. 09/882,076 filed June 15, 2001 (hereinafter, "this application").

2. I am aware that a Final Office Action was mailed in this application on April 21, 2005 and that, in this Final Office Action, all pending claims were rejected either as being obvious under 35 USC § 103(a) in view of U.S. Patent No. 6,829,713 B2 Cooper et al. (hereinafter, "Cooper") and further in view of U.S. Patent No. 6,625,688 B1 Fruehling et al. (hereinafter, "Frueling").

3. I am aware of an Amendment in the present application being filed in response to this subsequent Office Action and that this declaration is attached to that Amendment as Exhibit A.

4. The invention set forth in all claims submitted in this application, whether pending, original or previously amended, was conceived and actually reduced to practice by us in this country at least prior to December 30, 2000. Exhibit B, attached hereto, is a document that lists versions dating from September 3, 2000 to September 25, 2000 and describes adaptive throttling. The document includes a general description of adaptive throttling as well as including computer-executable algorithms (e.g., see algorithm on pages 8-9) that illustrate the constructive reduction to practice of the invention. This document therefore contains a dated description that evidences conception and reduction to practice of the claimed invention at least as early as September 3, 2000.

5. As further evidence of conception and reduction to practice Exhibit C, attached hereto, illustrates a redlined updated version of the document in exhibit B we made on October 29, 2000. As evidenced by the actual working computer-executable algorithms provided, this document further illustrates the conception and reduction to practice of the invention at least prior December 30, 2000.

6. We hereby declare that all statements made herein of our own knowledge are true and that all statements made on information and belief are believed to be true; and further that statements are made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such false statements may jeopardize the validity of the application or any patent issued thereon.

Date_____ _____
Stephane G. Plante

Date_____ _____
John D. Vert

Date 7-20-2005 Jacob Oshins
Jacob Oshins

EXHIBIT B

1.0 Document Overview

1.1 Document Purpose

The purpose of this paper is to describe a possible implementations for an Adaptive Throttling Policy. The intent of this document is to gather a permanent record of the design and thought process for patent and implementation verification purposes

1.2 Revision History

- V0.1, September 3rd, 2000. Initial Revision
- V0.2, September 6th, 2000. Per-Processor Performance State information
- V0.3, September 11th, 2000. Review comments merged
- V0.4, September 18th, 2000. Thermal Integration
- V0.9, September 19th, 2000. Battery Integration. Document is Complete.
- V0.91, September 25th, 2000. Typos & Corrections

2.0 Design Vision

2.1 What is Adaptive Throttling?

When a computer is running on batteries, it is not desirable to always run the CPU at its maximum available frequency. For example, if the computer is idle, because the user is reading a Microsoft Word document, running the CPU at full frequency merely drains the battery much more quickly.

Adaptive Throttling is the idea that the CPU should run at the maximum frequency required to fulfill the user's current needs. For example, while the user is reading the Microsoft Word document, the CPU should be throttled to its lowest possible frequency to save power. As soon as the user hits the page-down key or does anything else that requires the CPU, the CPU should be throttled back up to the frequency that places the CPU closest to being 100% busy as possible.

In practice, the system should only pick the highest-throttle for each of the voltage states supported. The reason being that if the CPU is idle enough of the time, it will spend a large portion of time in the C2 state, which effectively means that it has been throttled to the correct level. Since Microsoft has invested a great of energy into getting the C-State algorithms correct, this implementation should leverage that.

In the code, this is referred to as `PO_THROTTLE_ADAPTIVE`.

2.2 What is Degraded Throttling?

Degraded Throttling is a subset of Adaptive Throttling. The difference is that Adaptive Throttling does not put a cap on the maximum frequency that can be selected whereas Degraded Throttling does. This is useful in enforcing a policy where the user is willing to trade away some performance for longer battery life. It is particularly useful in situations

where the CPU is stuck "busy-waiting". Typically the Operating System should begin by placing the cap at the lower-voltage, highest-throttle state and decreasing to lower throttle states as battery capacity diminishes.

As an example, when the user hits the page-down key, the CPU might revert to 50% of its maximum frequency while the next part of the document is read from the disk and the screen is re-drawn. This might take a little longer than if the CPU had reverted to 100%, but it assumed that there would some saving in run the processor at a lower frequency for a longer period of time.

In the code, this is referred to as `PO_THROTTLE_DEGRADE`.

2.3 What is Constant Throttling?

Constant throttling is a subset of Adaptive Throttling and is very similar to Degraded Throttling. They both start at the lowest-voltage highest-frequency state, but unlike Degraded Throttling, Constant Throttling will never **force** the maximum throttle to goto a lower frequency state. The throttle is actually allowed to go to a lower frequency if so desired, but not forced to do so.

In the code, this is referred to as a `PO_THROTTLE_CONSTANT`.

2.4 Why Implement Adaptive Throttling?

We currently handle a few scenerios badly. And OEM perception, with the help of AMD and Intel, is that there are other scenarios that we handle badly which we think we handle well, which leads to them shipping their own drivers and crapplets.

- Machine is mostly or completely idle: We currently handle this well. We put the CPU into C3 via the SLP# signal. The CPU is as deeply asleep as it would be in S1.
- Machine is used to play Ms. PacMan, or any other app that eats some but not all of the CPU bandwidth. We currently handle this poorly. We put the CPU into C1 whenever we hit the idle loop, meaning that the CPU consumes quite a bit of power. Part of what makes this scenario tricky is that we don't know if the app will gracefully degrade if we take away CPU bandwidth. We should handle this scenario at first by trying to match CPU performance with CPU bandwidth. We can make the CPU bandwidth degrade over time if the Degrade policy is chosen.
- Machine is running with apps consuming all CPU. We handle this poorly. The CPU stays in C0. Out attitude in the past has been to find these apps and get them to fix it. Unfortunately, this attitude hasn't paid off.
- Machine is being used to play a DVD. We handle this poorly. The CPU stays in C0. The scenario is worthy of mention because it has special attributes. First, we know that restricting the CPU bandwidth will result in degraded performance, but the app will still run. Second, we could potentially find out how long the DVD is, allowing us to know how much total performance is needed. Third, on the laptops that we have looked at, DVD playback tends not to use all available CPU bandwidth.

2.5 Hardware Issues

Some concerns regarding the current and future generation of processors are important considerations.

- The hardware latency for changing the voltage is potentially long enough that it can cause CPU-availability problems. For example, a soft-modem can drop a connection if it isn't serviced every 10ms or that it will never make a connection if it isn't serviced every 2ms during the "training phase" (at the beginning of the call). Intel's best-case voltage switching time is around 2ms. Dell machines seem to take 24 retries, which brings them into the > 50ms range. AMD is currently at 200us, with an occasional retry. Transmeta claims to be at 20us
- The hardware latency for changing the CPU frequency without changing the voltage is around 3us. Unfortunately, we don't have a way of telling the kernel that the latency for entering a state is large if your're coming from a different voltage but tiny if you're staying with the same voltage. We could make this assumption directly in the kernel.
- CPU frequency is adjusted by causing the chipset to deassert the CLK_RUN# signal N out of M cycles, where M is usually 8. Deasserting CLK_RUN# is also what happens when we hit the C2 idle state
- Using the C3 idle state consumes significantly less power than the C2 power state. Throttling while using C3 causes the CPU to run longer, effectively putting it in C2 for part of the time that it could be in C3. This means that throttling while using C3 wastes power. IBM and others have shown us empirical data to support this.

2.6 Integration Issues

There are several minor issues that must be handled for the system to perform optimally.

- The system should return to highest-frequency highest-voltage when beginning any sort of power management operation (Sleep or Hibernate). This is essential during Hibernate to ensure that writing the hibernate file does not become a CPU-bound operation. It also insures that if the machine is transitioning to a Sleep or Hibernate state due to battery considerations, that the machine spends the minimal amount of time to make the transition. Just before entering the Sleep state, the processors should be returned to the lowest-voltage state possible.
- The implementation should respect the result of the thermal policy manager. If the thermal policy forces the throttle to be reduced, the Adaptive Throttling manager should not increase the throttle past that point
- The Operating System will perform better if we return to the lowest-voltage highest frequency state during any period of heavy C3 activity.

2.7 Time Management

For ease of integration with the existing Idle Promotion code base, all time units will be kept track of in terms of TickCounts. These are the units used in `Prcb->KernelTime`, `Prcb->UserTime`, and `Thread->KernelTime`. The following define will be used to establish the current system time:

```
#define CUR_TIME(X) (X->KernelTime + X->UserTime)
```

Where X represents a pointer to the current PRCB. The reason that we take in a pointer to the PRCB is that it is more efficient to get the PRCB once and then kept track of it in a local variable, even though KeGetCurrentPrcb() is expanded into an in-line call.

3.0 Data Structures

3.1 PROCESSOR_PERF_STATE

This data structured is defined in ntos\pop.h. This structure replaces (in the kernel only) PROCESSOR_PERF_LEVEL.

```
typedef struct {
    UCHAR      PercentFrequency;    // max = 100
    UCHAR      MinCapacity;         // Percentage
    USHORT     Power;               // milliwatts
    UCHAR      IncreaseLevel;       // goto higher freq
    UCHAR      DecreaseLevel;       // goto lower freq
    USHORT     Flags;               // Used for Flags
    ULONG      IncreaseTime;        // goto higher freq
    ULONG      DecreaseTime;        // goto lower freq
    ULONG      IncreaseCount;       // goto higher freq
    ULONG      DecreaseCount;       // goto lower freq
    ULONGLONG PerformanceTime;      // for tick count
} PROCESSOR_PERF_STATE, *PPROCESSOR_PERF_STATE
```

The kernel will allocate an array of these structures and store the pointer to the array in the Processor's PRCB.

The following Flags are defined as being available:

```
#define POP_THROTTLE_NON_LINEAR          0x1
```

PercentFrequency is the normalized representation of frequency that this performance state represents. The highest performance state has a frequency of 100%, if it is available. This avoids the problem of dealing with faster and faster CPUs. Under this mechanism, a CPU that has a max speed of 450MHz uses the same algorithm as one that runs at 700Mhz.

MinCapacity is used to represent the minimum battery remaining capacity that is required for the CPU to be in this state. It should be noted that this only applies if the machine is running on DC since the relevant throttling policies are only available then. This value is expressed as a percentage.

IncreaseLevel and DecreaseLevel are the boundaries of the bucket that defines the current state. If the CPU is busier than IncreaseLevel, the Operating System should pick a higher processor frequency. If the CPU is less busy than DecreaseLevel, the OperatingSystem should pick a lower processor frequency.

It should be noted that `PercentFrequency` will be higher than `IncreaseLevel` since the only way to reach `PercentFrequency` is for the Operating System to be 100% busy for the given frequency. That is, the CPU must be using every single cycle allocated to it in order to be running at `PercentFrequency` level of business. In order to allow for promotion in cases where the system is not quite at that level of business, `IncreaseLevel` must be a smaller value than `PercentFrequency`. If `IncreaseLevel` is higher than `PercentFrequency`, then promotion can never occur.

`IncreaseCount` and `DecreaseCount` are used to keep track of the number of transitions **from** this state to another performance state. Transitions to a lower performance state cause an increase in `IncreaseCount`. Transitions to a higher performance state cause an increase in `DecreaseCount`.

`PerformanceTime` is used to keep trace of the number of ticks that are spent at this performance level. This value is updated whenever the processor switches to a different performance state. When the user queries the performance information, the current elapsed time at this state is added to `PerformanceTime` for the current performance state.

3.2 PROCESSOR_POWER_STATE

This structure is defined in sdkinc\ntpoacpi.h. This structure already exists but must be grown to accommodate more information. Changed elements are listed in red.

```
typedef struct {
    PPROCESSOR_IDLE_FUNCTION IdleFunction;
    ULONG                    Idle0KernelTimeLimit;
    ULONG                    Idle0LastTime;
    PVOID                    IdleState;
    ULONGLONG                LastCheck;
    PROCESSOR_IDLE_TIMES     IdleTimes;
    ULONG                    IdleTime1;
    ULONG                    PromotionCheck;
    ULONG                    IdleTime2;
    UCHAR                    CurrentThrottle;
    UCHAR                    ThermalThrottleLimit;
    UCHAR                    Spare1[2];
    UCHAR                    ThermalThrottleIndex;
    UCHAR                    CurrentThrottleIndex;
    ULONG                    Spare2[2];
    ULONG                    PerfSystemTime;
    ULONG                    PerfIdleTime;
    // Temp for debugging…
    ULONGLONG                DebugDelta;
    ULONG                    DebugCount;
    ULONG                    LastSysTime;
    ULONGLONG                TotalIdleStateTime[3];
```

```
    ULONG                           TotalIdleTransitions[3];
    ULONG                           Spare3;
    ULONGLONG                       PreviousC3StateTime;
    UCHAR                           KneeThrottleIndex;
    UCHAR                           ThrottleLimitIndex;
    UCHAR                           PerfStatesCount;
    UCHAR                           Spare1[1];
    ULONG                           Flags;
    LARGE_INTEGER                   PerfCounterFrequency;
    ULONG                           PerfTickCount;
    KTIMER                          PerfTimer;
    KDPC                            PerfDpc;
    PPROCESSOR_PERF_STATE           PerfStates;
    PSET_PROCESSOR_THROTTLE         PerfSetThrottle;
} PROCESSOR_POWER_STATE, *PPROCESSOR_POWER_STATE;
```

The `PerfDpc` and `PerfTimer` variables are convenient storage areas for the context information that will be required to fire a periodic timer to make sure that the CPU is not too busy for its current performance level. The `Flags` field will be used to store useful state information. The only useful defines (as of this document) are:

```
#define PSTATE_ADAPTIVE_THROTTLE    0x1
#define PSTATE_DEGRADED_THROTTLE    0x2
#define PSTATE_CONSTANT_THROTTLE    0x4
```

It should be noted that `PSTATE_ADAPTIVE_THROTTLE` is what turns on the entire throttling behavior and that the other flags are used to modify the behavior. `PSTATE_DEGRADED_THROTTLE` and `PSTATE_CONSTANT_THROTTLE` are also mutually exclusive flags.

The `PerfSystemTime` and `PerfIdleTime` structure are used to keep track of previous values to calculate the important time deltas. `PerfSystemTime` is used to store the amount of system time previously elapsed, as expressed by CUR_TIME(`Prcb`). `PerfIdleTime` is used to store the amount of time that was spent in the idle thread as expressed by `Thread->KernelTime`.

`PerfStates` is a pointer to the `PROCESSOR_PERF_LEVEL` array associated with this processor. Each processor must have unique copy of this structure to maintain per-processor information about the amount of time spent in each state. `PerfStatesCount` is the number of elements within this array.

`CurrentThrottleIndex` is the index in the `PerfStates` array that has currently been selected. This has been done using a `for` loop to find the entry in the `PerfStates` array that corresponds to `CurrentThrottle`.

`KneeThrottleIndex` is the index in the `PerfStates` array that represents the lowest-voltage highest frequency part of the curve. This value is pre-calculated since the Degraded and Constant Throttling policies depend upon it.

`ThrottleLimitIndex` is the index in the `PerfStates` array that represents the maximum state that is acceptable under the current policy. The maximum of this value is the `KneeThrottleIndex`. This value is modified by the Kernel's Battery subsystem whenever it receives new battery capacity remaining information.

`ThermalThrottleIndex` is the index in the `PerfStates` array that represents the maximum state that is acceptable based upon the thermal throttle. This value is used under all policies.

`PreviousC3StateTime` was the amount of time spent at C3 during the last successful throttle check. We need this time to compute the delta amount of time spent at C3 during the previous interval and thus determine the percentage time we spent at C3.

`PerfTickCount` is the `CUR_TIME(Prcb)` when the processor switches to a new performance state. When the processor switches to a new performance state, the `CUR_TIME(Prcb)` minus `PerfTickCount` is added to the `PerformanceTime` bucket for the previous performance level.

`PerfCounterFrequency` is the stored value of `KeQueryPerformanceCounter()` to obtain the frequency rating of the counters. This is required since we want to minimize the number of calls we make and it's a good idea to cache this number. `PerfCounterFrequency` is used to make the transitions between values which are stored in PerformanceCounter units and those that are stored in Tick Counts.

`PerfSetThrottle` is the function that will get called when the Operating System wants to set a new throttle. The reason that this is in this structure instead of a global variable is that we want to properly synchronize access to it.

4.0 Algorithms

4.1 Setting a Throttle Level

The following code will be used to set a particular processor to a specific thermal level

```
VOID
FASTCALL
PopSetThrottle(
      PPROCESSOR_POWER_STATE  PState,
      PPROCESSOR_PERF_STATE   PerfStates,
      ULONG                   Index,
      ULONG                   SystemTime,
      ULONG                   IdleTime
      )
{
      NTSTATUS    Status;
      UCHAR       Current = PState->CurrentThrottleIndex;
```

```
    //
    // Actually set the processor to the new throttle level
    //
    Status = PState->PerfSetThrottle(
          PerfStates[Index].PercentFrequency
          );
    if (!NT_SUCCESS(Status)) {

          //
          // If it didn't succeed, then don't update the
          // stats.
          //
          return;

    }

    //
    // Update the booking for the current state
    //
    PerfStates[Current].PerformanceTime += (SystemTime -
          PState->PerfTickCount);

    //
    // Update the current throttle information
    //
    PState->CurrentThrottle = PerfStates[Index].PercentFrequency;
    PState->CurrentThrottleIndex = Index;

    //
          // Update our idea of what the current tick counts are
          //
          PState->PerfIdleTime = IdleTime;
          PState->PerfSystemTime = SystemTime;
          PState->PerfTickCount = SystemCount;

          //
          // Remember how much we spent in C3 at this point
          //
          PState->PreviousC3StateTime = PState->TotalIdleState[2];
}
```

This function, which will be called only on the target processor, while running either at DPC level or within the affinity of the target processor, will actually set the new throttle and update the bookkeeping.

4.2 Busy and C3 Detection

The following code can be called within the context of the target processor to determine how busy the CPU has been during the previous time period. This function would typically be called from the IdleThread, a DPC, or while running at DISPATCH_LEVEL.

```
UCHAR
CalculateBusyPercentage(
      PPROCESSOR_POWER_STATE  PState
      )
{
```

```
        PKPRCB      Prcb;
        PKTHREAD    Thread;
        UCHAR       Frequency;
        ULONGLONG   Idle;
        ULONG       Busy;
        ULONG       IdleTimeDelta;
        ULONG       CpuTimeDelta;

        Thread = KeGetCurrentThread();
        Prcb = CONTAINING_RECORD( PState, KPRCB, PowerState);

        IdleTimeDelta = Thread->KernelTime - PState->PerfIdleTime;
        CpuTimeDelta = CUR_TIME(Prcb) - PState->PerfSystemTime;
        Idle = (IdleTimeDelta * 100)/(CpuTimeDelta);

        //
        // We cannot be more than 100% idle, and if we are then we
        // are 0% busy (by definition), so apply the proper caps
        //
        if (Idle > 100) {

              Return 0;

        }
        Busy = 100 - Idle;
        Frequency = (UCHAR) (Busy * PState->CurrentThrottle /
              POWER_PERF_SCALE);
        return Frequency;
}
```

The Idle and Busy values represent a percentage of what the CPU was doing during the last interval. To simplify the math later on, these numbers are normalized against the current throttle value.

For example:

- If the CPU was 50% busy at 50%Throttle, that really means that the CPU was 25% busy at 100% throttle.
- If the CPU was 100% busy at 25% Throttle, that really means that the CPU was 25% busy at 100% throttle.
- If the CPU was 10% busy at 10% Throttle, that really means that the CPU was 1% busy at 100% throttle.

Similarly, the formula for detecting how much time the CPU has spent in C3 during a known interval is:

```
UCHAR
CalculateC3Percentage(
      PPROCESSOR_POWER_STATE  PState
      )
{
      PKPRCB            Prcb;
      ULONGLONG         CpuTimeDelta;
      ULONGLONG         C3;
```

```
LARGE_INTEGER     C3Delta;

Prcb = CONTAINING_RECORD( PState, KPRCB, PowerState);

//
// Calculate the C3 time delta in terms of Nanosecs.
// The formulas for conversion are taken from
// PopConvertUsToPerfCount
//
C3Delta.QuadPart = PState->TotalIdleState[2] -
      PState->PreviousC3StateTime;
C3Delta.QuadPart = (US2SEC * US2TIME * C3Delta.QuadPart) /
      PState->PerfCounterFrequency.QuadPart;

//
// Now, calculate the CpuTimeDelta in terms of
// NanoSeconds
//
CpuTimeDelta = (CUR_TIME(PRCB) - PState->PerfSystemTime) *
      KeTimeIncrement;

//
// Figure out the ratio of the two, and cap it
// at 100%
//
C3 = C3Delta.QuadPart * 100 / CpuTimeDelta;
If (C3 > 100) {
      Return 100;
      }
      return (UCHAR) C3;
}
```

4.3 Calculating IncreaseLevel

To calculate what the upper bound for any PROCESSOR_PERF_STATE, the following rules apply:

```
VOID
CalculateIncreaseLevel(
      PPROCESSOR_PERF_STATE   CpuState,
      ULONG                   CpuStateCount
      )
{
      ULONG I;
      ULONG DeltaPerf;

      //
      // Optimization for case where there are no CpuStates
      //
      if (CpuStateCount == 0) {

            return;

      }

      //
```

```
        // This guarantees that we can never promote past this state
        //
        CpuState[0].IncreaseLevel = CpuState[0].PercentFrequency + 1;

        //
        // Calculate the increase Level
        //
        For (I=1; I < CpuStateCount;I++) {

              DeltaPerf = CpuState[I-1].PercentFreqency -
                    CpuState[I].PercentFreqency;
              DeltaPerf *= PopPerfIncreasePercentModifier;
              DeltaPerf /= POWER_PERF_SCALE;
              DeltaPerf += PopPerfIncreaseAbsoluteModifier;
              If (DeltaPerf > CpuState[I].PercentFrequency) {

                    DeltaPerf = POWER_PERF_SCALE;

              } else {

                    DeltaPerf = CpuState[I].PercentFrequency -
                          DeltaPerf;

              }
              CpuState[I].IncreaseLevel = (UCHAR) DeltaPerf;
        }
}
```

It should be noted that the increase level will always result in the percentage business required for a promotion to the next higher throttle level, regardless of whether or not a voltage change is required. The reason that this is the case is because it is impossible to actually increase more than one level using the Idle Detection algorithm previously presented.

If its not desired that we should promote to a higher frequency within the same voltage band, than this could be accomplished by removing any non-linear states from the list of states or by forcing the increase level to be the same value used by the highest frequency state in the voltage range.

4.4 Calculating DecreaseLevel

To calculate what the lower bound for any PROCESSOR_PWER_STATE, the following rules apply:

```
VOID
CalculateDecreaseLevel(
      PPROCESSOR_PERF_STATE   CpuState,
      ULONG                   CpuStateCount
      )
{
      //
      // We will be required to walk the CpuState array several times
      // and the only way to safely keep track of which index we are
      // looking at versus the one we care about is to use to variable
```

```
// to keep track of indexes.
//
ULONG I, J;
ULONG DeltaPerf;

//
// Calculate the decrease level
//
for (I=0;I < CpuStateCount;I++) {

      if (I == (CpuStateCount-1)) {

            CpuState[I].DecreaseLevel = 0;
            Continue;

      }

      DeltaPerf = CpuState[I-1].PercentFrequency -
            CpuState[I].PercentFrequency;
      DeltaPerf *= PopPerfDecreasePercentModifier;
      DeltaPerf /= POWER_PERF_SCALE;
      DeltaPerf += PopPerfDecreaseAbsoluteModifier;

      if (DeltaPerf > CpuState[I].PercentFrequency) {

            DeltaPerf = 0;

      } else {

            DeltaPerf = CpuState[I].PercentFrequency - DeltaPerf;

      }
      CpuState[I].DecreaseLevel = (UCHAR) DeltaPerf;
}

//
// We want to eliminate demotions at the same voltage
// level, so guarantee that the decrease levels result
// in being set to the next voltage level...
//
I = 0;
while ( I < CpuStateCount ) {

      //
      // Find the next non-linear state. We assume that I
      // is currently pointing at the highest-frequency
      // state within a voltage band and we are interesting
      // in finding the highest-frequency state at the
      // next-lower voltage band.
      //
      for (J = I + 1; J < CpuStateCount; J++ ) {

            If (CpuState[J]->Flags & POP_THROTTLE_NON_LINEAR) {

                  Break;

            }
```

```
}

//
// Want to find the previous state since that
// will be the decrease limit that we will use
//
J--;

//
// Set the decrease limit to this new level
//
while (I < J) {

      CpuState[I]->DecreaseLevel =
            CpuState[J]->DecreaseLevel;
      I++;

}

//
// Skip the Jth state since it is the bottom of
// the frequencies available for the current
// voltage level. Note that we are skipping this
// from I's point of view.
//
I++;
      }

}
```

This algorithm looks at the list of available states twice. The first time, it calculates the value that would be used to decrease the throttle to the next lower value. The second time, it calculate the value that would be used to decrease the throttle to the next lower non-linear state. The reason that this algorithm is used is because there are almost power savings to decreasing the frequency but keeping the voltage constant. Thus, any demotions should result in voltage changes.

The reason that decreasing the frequency but keeping the voltage produces few power savings is that with an aggressive C2 policy, running the CPU at a higher frequency while spending lots of time in C2 is equivalent to running the CPU at a lower frequency while spending no time in C2.

4.5 Calculting Increase/DecreaseTime

```
VOID
CalculateIncreaseDecreaseTime(
      PPROCESSOR_PERF_STATE         CpuState,
      ULONG                         CpuStateCount,
      PPROCESSOR_STATE_HANDLER2     PerfHandler
      )
{
      ULONG I;
```

```
If (CpuStateCount == 0) {
      Return;
}

//
// We can never increase from State 0
//
CpuState[0].IncreaseTime = (ULONG) -1;

//
// Loop over the next elements...
//
For (I = 1; I < CpuStateCount; I++ ) {

      //
      // Decrease Time of previous state
      // should be based on whether current state is
      // linear or not
      //
      CpuState[I-1].DecreaseTime =
            PerfHandler->HardwareLatency * 10 *
            PopPerfDecreaseTimeValue;
      If (CpuState[I].Flags & POP_THROTTLE_NON_LINEAR)
      {
            CpuState[I-1].DecreaseTime *= 10;
      }

      //
      // Increase Time of current state should be
      // based on whether or not the state is
      // linear or not
      //
      CpuState[I].IncreaseTime =
            PerfHandler->HardwareLatency * 10 *
            PopPerfIncreaseTimeValue;
      If (CpuState[I-1].Flags &
            POP_THROTTLE_NON_LINEAR)
      {
            CpuState[I].IncreaseTime *= 10;
      }
}

//
// We can never decrease from the last state
//
I--;
CpuState[I].DecreaseTime = (ULONG) -1;

}
```

4.6 Calculate MinCapacity

This routine is used to determine at what levels of battery capacity we start to force throtte when we are running on the degraded throttle.

```
VOID
CalculateMinCapacity(
      PPROCESSOR_PERF_STATE   PerfStates,
      ULONG                   PerfStatesCount,
      PPROCESSOR_POWER_STATE  PState
)
{
      UCHAR I;
      UCHAR Num;
      UCHAR Total = PopPerfDegradeThrottleMinCapacity;
      UCHAR Width = 0;

      if (!PerfStatesCount) {

            return;

      }

      for (I = 0; I < PState->KneeThrottleIndex; I++) {

            //
            // Any of the steps before the knee are set to 100%
            //
            PerfStates[I].MinCapacity = 100;

      }

      //
// Calculate the range for which we clamp down the throttle
//
Num = PerfStatesCount - (PState->KneeThrottleIndex + 1);
if (Num != 0) {

      Width = Total / Num;

}

//
// Look at all the states from the Knee to the end. Starting
// at the highest value, set the min capacity and subtract the
// appropriate value to get the next min capacity.
//
for (I = PState->KneeThrottleIndex; I < PerfStatesCount; I++) {

      //
      // We put a floor onto how low we can force the throttle
      // down to. If this state is operating below that floor,
      // then we should set the MinCapacity to 0, which
      // reflects the fact that we don't want to degrade
      // past this point
      //
      if (PState[I].PercentFrequency <
            PopPerfDegradeThrottleMinFrequency) {

            //
            // We modify the min capacity for the
            // previous state since we don't ever
```

```
                        // want to demote from that state.
                        // Also, once we start being less than
                        // the min frequency, the min capacity
                        // will always be set to 0, except for
                        // the last state. But this is okay since
                        // we look at each state in order. We also
                        // have to make sure that violate array
                        // bounds, but this can only happen if
                        // the perf states array is badly formed
                        // or the min frequency is badly formed.
                        //
                        if (I != 0) {

                              PerfStates[I-1].MinCapacity = 0;

                        }

                  }

                  PState[I].MinCapacity = Total;
                  Total -= Width;

            }
      }
```

The logic behind this algorithm is that it is designed to allow the system to run at Lowest Voltage-Highest Frequency (henceforth the KneeState) while the battery capacity remaining is between 100% and `PopPerfDegradeThrottleMinCapacity`. Once the battery capacity falls below that level, the highest allowed state is the next-Highest Frequency available at the same voltage. When the battery capacity degrades past a certain amount (which is based on the `PopPerfDegradeThrottleMinCapacity` and the number of available states), the highest allowed state becomes the next-Highest Frequency remaining. Thus, as the battery capacity diminishes, the processor runs at a lower and lower frequency.

Another feature of this algorithm is that it allows for the OS to specify what the smallest frequency that the system can be forced to degrade to. For example, if a system supports the following frequencies at the low voltage state: 50%, 40%, 30%, 20%, and 10%, then by setting `PopPerfDegradeThrottleMinFrequency` to 30% will guarantee that we will degrade the throttle below 30%.

To minimize the amount of calculations we must make at run time, we can pre-calculate the minimum battery capacity that must be remaining to be in the indicated performance state. It should noted that any algorithm used must be able to handle where there are no more performance states after the knee state.

The beauty of this algorithm is that is can be easily replaced. If the Performance team decides that a log, exp, or any other means of calculating the "curve" is more appropriate, then only this function needs to be changed.

A sample equation could be:

$$\min Capacity = \frac{thresholdcapacity*(currentFrequency^2 - \min Frequency^2)}{(\max Frequency^2 - \min Frequency^2)}$$

4.7 System IdleLoop

To prevent the system from considering the promotion and demotion of performance states as part of how busy the system is, it is clearly desirable to invoke the increase and decrease of the CPU throttles from within the idle loop.

The basic idea of the code in the idle loop is that the code should check to see what percentage (normalized to `POP_PERF_SCALE`) of time during the last interval was spent running the idle thread and how much time was spent doing actual work. Once the percentage of actual work done is calculated, then the Operating System should look at the `PerfStates` table to see if this value falls within the operating parameters for the current bucket. It is important to make this check first because buckets are allowed to overlap each other.

If the value does not fall within the current bucket, then the operating system finds the closest performance state for which the value matches the parameters of the bucket. There is an important distinction here since by picking the nearest bucket, the operating system will have a tendency to not pick the highest or lower performance states unless there is absolutely no other choice. The advantage of this algorithm is that it will pick the `PercentFrequency` closest to the calculated value.

It is important to note that the Idle loop runs at DPC level and within the context of the processor for which it is targeted. That means that the code within the Idle loop cannot call anything that is marked as pageable. The benefit of running at DPC level is that no synchronization is required if the data structures used by the Idle loop can only be accessed by a thread running on the same processor at DPC level. This function should also be called before the C-State handler is invoked.

The algorithm would look something like this:

```
VOID
PoPerfIdle(
      PPROCESSOR_POWER_STATE  PState;
      )
{
      BOOLEAN                 C3Forced = FALSE;
      BOOLEAN                 Promoted = FALSE;
      BOOLEAN                 Demoted = FALSE;
      KPRCB                   Prcb;
      PPROCESSOR_PERF_STATE   PerfStates;
      ULONG                   TickCount;
      UCHAR                   I;
      UCHAR                   J;
      UCHAR                   CurrentPerfState
      UCHAR                   Freq;
      ULONG                   IdleTime;
      ULONG                   Time;
      ULONG                   TimeDelta;
```

```
ULONG                       PerfStatesCount;

//
// This piece of code should actually be done in the main
// PopIdle0 or PopProcessorIdle routines to save a function
// call. However this code is included here for completeness.
//
Prcb = CONTAINING_RECORD( PState, KPRCB, PowerState );
if (!PState->Flags & PSTATE_ADAPTIVE_THROTTLE) {
      return;
}

//
// Has enough time expired?
//
Time = CUR_TIME(Prcb);
IdleTime = Thread->KernelTime;
TimeDelta = Time - PState->PerfSystemTime;
if (TimeDelta < PopPerfTimeDelta) {
      return;
}

//
// Remember what the perf states are
//
PerfStates = PState->PerfStates;
PerfStatesCount = PState->PerfStatesCount;

//
// Find the bucket with the correct frequency
//
CurrentPerfState = PState->CurrentThrottleIndex;
I = CurrentPerfState;

//
// At this point, we need to see if the number of C3
// transitions have exceeded a threshold value, and if
// so, then we really need to throttle back to the
// KneeThrottleIndex since we save more power if the
// processor is at 100% and in C3 than if the processor
// is at 12.5% and in C3.
//
Freq = CalculateC3Frequency(PState);
If (Freq >= PopPerfMaxC3Frequency) {

      //
      // Set the throttle to the lowest knee in
      // the voltage & frequency curve
      //
      I = PState->KneeThrottleIndex;
      if (CurrentPerfState > I) {

            Promoted = TRUE;

      } else if (CurrentPerfState < I) {

            Demoted = TRUE;
```

```
            }

            //
            // Remember why we are doing this
            //
            C3Forced = TRUE;

            //
            // Skip to setting the throttle
            //
            goto PoPerfIdleSetThrottle;

      }

      //
      // Calculate how busy the CPU is
      //
      Freq = CalculateIdleFrequency(PState);

      //
      // Have we exceeded the thermal throttle limit?
      //
      If (Freq > PState->ThermalThrottleLimit) {

            //
            // The following code will force the frequency to
            // only as busy as the Thermal Throttle Limit will
            // actually allow. This removes the need for complicated
            // algorithms later on.
            //
            Freq = PState->ThermalThrottleLimit;
            I = PState->ThermalThrottleIndex;

      }

      //
      // Is there an upper limit to what the throttle can goto?
      // Note that because we check these after we have checked
      // the thermal limit, it means that it is not possible for
      // frequency to exceed the thermal limit that we have specified
      //
      if (PState->Flags & PSTATE_DEGRADED_THROTTLE) {

            //
            // Make sure that we don't exceed the
            // state that is specified
            //
            J = PState->ThrottleLimitIndex;
            If (Freq >= PerfStates[J].PercentFrequency) {

                  Freq = PerfStates[J].IncreaseLevel;
                  I = J;

            }

      } else if (PState->Flags & PSTATE_CONSTANT_THROTTLE) {
```

```
            J = PState->KneeThrottleIndex;
            If (Freq >= PerfStates[J].PercentFrequency) {

                  Freq = PerfStates[J].IncreaseLevel;
                  I = J;

            }

      }

      //
      // Find the processor frequency that best matches
      // the one that we have just calculated. Please
      // note that this algorithm is written in such
      // a way that I can only travel in a single
      // direction. It is possible to collapse the
      // following code down, but not without allowing
      // the possibility of I doing a "yo-yo" between
      // two states (and thus never terminating the
      // while-loop).
      //
      if (PerfState[I].IncreaseLevel < Freq) {

            If (I != 0) {

                  Promoted = TRUE;
                  I--;

            }

      } else if (PerfStates[I].DecreaseLevel > Freq) {

            while (1) {

                  If (I==(PerfStatesCount-1)) {

                        // don't exceed the array
                        break;

                  }
                  Demoted = TRUE;
                  I++;
                  If (PerfStates[I].DecreaseLevel <= Freq) {

                        break;

                  }

            }

      }

PoPerfIdleSetThrottle:

      //
      // Note we need to do this now because we don't want
```

```
// to exit this code path without having set or cancelled
// the timer as is appropriate. The only exception to
// this rule is in the case where the system hit the
// C3 limit.
//
// Cancel the timer under the following conditions
//
if (I == 0) {

      //
      // We are at 100% throttle, so timer won't
      // do much of anything
      //
      KeCancelTimer(&(PState->PerfTimer));

} else if (PState->Flags & PSTATE_CONSTANT_THROTTLE &&
      I == PState->KneeLimitIndex) {

      //
      // We are at the maximum throttle allowed
      //
      KeCancelTimer(&(PState->PerfTimer));

} else if (PState->Flags & PSTATE_DEGRADED_THROTTLE &&
      I == PState->ThrottleLimitIndex) {

      //
      // We are at the maximum throttle allowed
      //
      KeCancelTimer(&(PState->PerfTimer));

} else {

      //
      // No restrictions that we can think of,
      // so set the timer. Note that the semantics
      // of KeSetTimer are useful here --- if the
      // timer has already been set, then this
      // resets it (moves it to the non-signaled
      // state) and recomputes the period.
      //
      KeSetTimer(
            &PState->PerfTimer,
            …,
            &Pstate->PerfGpe
            );

}

//
// We have to make special allowances if we were forced to
// throttle because of C3 considerations
//
if (!C3Forced) {

      //
      // See if enough time has expired to justify changing
```

```
            // the throttle. This code is here because certain
            // transitions are fairly expensive (like those across a
            // voltage state)while others are cheap. So the amount of
            // time required before we will consider promotion/demotion
            // from the expensive state might be longer than the
            // interval at which we run this function.
            //
            if ((Promoted && TimeDelta < PerfStates[I].IncreaseTime) ||
                (Demoted && TimeDelta < PerfStates[I].DecreaseTime)) {

                  //
                  // We haven't had enough time in the current
                  // state to justify the promotion or demotion.
                  // We don't update the bookkeeping since we
                  // haven't considered the current interval as
                  // as "success". So, we just return.
                  //
                  // N.B. It is very important that we don't
                  // update PState->PerfSystemTime here. If we
                  // did, then it is possible that TimeDelta would
                  // never exceed the thresholds required.
                  //
                  return;

            }

      }

      //
      // At this point, we need to update the bookkeeping
      //
      PState->PerfIdleTime = IdleTime;
      PState->PerfSystemTime = Time;
      PState->PreviousC3StateTime = PState->TotalIdleState[2];

      //
      // Update the promote and demote count
      //
      if (Promoted) {

            PerfStates[CurrentPerfState].IncreaseCount++;

      } else if (Demoted) {

            PerfStates[CurrentPerfState].DecreaseCount++;

      } else {

            //
            // At this point, we realize that we aren't
            // promoting or demoting and all the bookkeeping
            // is in order, so the appropriate thing to do
            // is just return.
            //
            return;

      }
```

```
        //
        // We have a new throttle. Update the bookkeeping to
        // reflect the amount of time that we spent in the
        // previous state and reset the count for the next
        // state
        //
        PopSetThrottle(
              PState,
              PerfStates,
              I,
              Time,
              IdleTime
              );
}
```

4.8 Processor Perf DPC

A desirable feature to have in an adaptive throttling mechanism is the ability to sense that the CPU has become 100% and that the throttle should be increased if required. The way to accomplish this is to schedule a periodic timer that fires if the throttle is not set to 100%.

It is important to note that this DPC may fire in situations where the CPU is not 100% busy within a given time quantum. However, since the Idle Handler resets the timer count every time it runs, the number of spurious calls to this routine should be small.

It is important to note that once the DPC has been fired, there is no need to cancel the timer since it is not scheduled as a periodic timer.

The sample algorithm would look like this:

```
VOID
PopPerfAdaptiveThrottleDpc(
      IN    PKDPC Dpc,
      IN    PVOID DpcContext,
      IN    PVOID SystemArgument1,
      IN    PVOID SystemArgument2
      )
{
      PKPRCB                    Prcb;
      PKTHREAD                  IdleThread;
      PPROCESSOR_PERF_STATE     PerfStates;
      PPROCESSOR_POWER_STATE    PState;
      UCHAR                     CurrentPerfState;
      UCHAR                     I;
      ULONG                     IdleTime;
      ULONG                     Time;
      ULONG                     TimeDelta;

      //
      // We need to fetch the PRCB and the PState structures.
      // We could easily call KeGetCurrentPrcb() here, but since
      // had room for a single context, why bother making the
```

```
// inline call (which generates more code than using the
// context field anyways) when we can simply remember it.
// The memory for the context field is already allocated
// anyways.
//
Prcb = (PKPRCB) DpcContext;
PState = &(Prcb->PowerState);

//
// Remember what the perf states are
//
PerfStates = PState->PerfStates;
CurrentPerfState = PState->CurrentThrottleIndex;

//
// Lets see if enough kernel time has expired since
// the last check...
//
Time = CUR_TIME(Prcb);
TimeDelta = Time - Pstate->PerfSystemTime;
if (TimeDelta < PopPerfCriticalTimeDelta) {
      return;
}

//
// How much time has expired on the Idle thread?
//
IdleThread = Prcb->IdleThread;
IdleTime = IdleThread->KernelTime;
TimeDelta = IdleTime - PState->PerfIdleTime;
if (TimeDelta < PopPerfCriticalIdleTimeDelta) {
      return;
}

//
// At this point, we think that the idle thread is
// stalled and that we need to do something fast to
// get it moving... Like setting it to the perf state
// that corresponds to the "knee" of the graph
// or a perf state higher than the current one...
//
if (PState->Flags & PSTATE_CONSTANT_THROTTLE) {

      //
      // Pick the knee of the curve
      //
      I = PState->KneeThrottleIndex;

} else if (PState->Flags & PSTATE_DEGRADED_THROTTLE) {

      //
      // Pick the maximum that we are allowed to goto
      //
      I = PState->ThrottleLimitIndex;

} else {
```

```
                //
                // Goto 100%
                //
                I = 0;

        }

        //
        // Set the new throttle
        //
        PopSetThrottle(
                PState,
                PerfStates,
                I,
                Time,
                IdleTime
                );
}
```

5.0 Initialization Changes

5.1 Global Variable Initialization

The following global variables must be initialized at the same time that the kernel is loaded. To simplify changing these values later on, the Kernel will provide some default values that can be overridden by the registry.

- PopPerfTimeDelta: A value in the same units as PRCB->KernelTime that corresponds to the Time Delta that must have occurred before the Idle thread will attempt to determine how busy the CPU was during the previous interval.
- PopPerfCriticalTimeDelta: A value in the same units as PRCB->KernelTime that corresponds to the Time Delta that must have occurred before the Timer DPC will attempt to determine how busy the CPU was during the previous interval
- PopPerfCriticalIdleTimeDelta: A value in the same units of PRCB->KernelTime that corresponds to the Time Delta that must have occurred for the IdleThread during a PopPerfCriticalTimeDelta period. If this time has not occurred, then the throttle will be raised by the TimerDPC.
- PopPerfIncreasePercentModifier: A value between 0 and 100 where lower means that overall IncreaseLevel value will be higher (and thus promotions won't occur as frequently) that indicates what percentage of the delta between the current state and the state to promote to should be used to set the promote level. A suggested value would be 0%
- PopPerfIncreaseAbsoluteModifier: A value between 0 and 100 where lower means that the overall IncreaseLevel value will be higher (and thus promotions won't occur as frequently) that indicates how many extra percentage points to remove to the promote level. It should be noted that if the value is particularly high, then it might not be possible to promote from this state. A suggested value would be 1%.
- PopPerfDecreasePercentModifier: A value between 0 and 100 where higher means that overall DecreaseLevel value will be lower (and thus demotions won't

occur as frequently) that indicates what percentage of the delta between the current state and the state to demote to should be used to set the demote level. A suggested value would be 50%

- PopPerfDecreaseAbsoluteModifier: A value between 0 and 100 where higher means that the overall DecreaseLevel value will be lower (and thus demotions won't occur as frequently) that indicates how many extra percentage points to subtract to the demote level. It should be noted that if the value is particularly high, then it might not be possible to demote from this state. A suggested value would be 1%.
- PopPerfIncreaseTimeValue: A value in the same units as PRCB->KernelTime that corresponds to the Time Delta that must have occurred before a Throttle Increase is considered. This value should be in multiple of PopPerfTimeDelta. This value may also serve a basis for calculating different time increments for each Throttle Step.
- PopPerfDecreaseTimeValue: A value in the same units as PRCB->KernelTime that corresponds to the Time Delta that must have occurred before a Throttle Decrease is considered. This value should be in multiple of PopPerfTimeDelta. This value may also serve a basis for calculating different time increments for each Throttle Step.
- PopProcPerfStateLookAsideList: This is a lookaside list that will be used to allocate `PROCESSOR_PERF_STATE` structures.
- PopPerfDegradeThrottleMinCapacity: A value between 0 and 100 that represents at what point of battery capacity we will start forcing down the throttle when we are in the Degraded Throttling mode. For example, a value of 50% means that we will start throttling when the CPU reaches 50%.
- PopPerfDegradeThrottleMinFrequency: A value between 0 and 100 that represents the lowest frequency that we can force the throttle down to when we are start the Degraded Throttling mode. For example, a value of 30% means that we will force the throttle below 30%.

5.2 PoInitializePrcb()

The PROCESSOR_POWER_STATE structure is initialized by PoInitializePrcb(). The following changes must occur:

```
VOID
FASTCALL
PoInitializePrcb(
      PKPRCB      Prcb
      )
{
      //
      // Zero power state structure
      //
      RtlZeroMemory(&Prcb->PowerState,sizeof(Prcb->PowerState));

      //
      // Initialize to legacy functions with promotion from it disabled
      //
      Prcb->PowerState.Idle0KernelTimeLimit = (ULONG) -1;
```

```
        Prcb->PowerState.IdleFunction = PopIdle0;
        Prcb->PowerState.CurrentThrottle = POP_PERF_SCALE;

        //
        // Initialize the Adaptive throttling subcomponents
        //
        KeInitializeDpc(
              &(Prcb->PowerState.PerfDpc),
              PopPerfAdaptiveThrottleDpc,
              NULL
              );
        KeSetTargetProcessorDpc(
              &(Prcb->PowerState.PerfDpc),
              Prcb->Number
              );
        KeInitalizeTimer(
              &(Prcb->PowerState.PerfTimer)
              );
}
```

5.3 PopSetPerfLevels()

The PROCESSOR_PERF_STATE structure is initialized by calls to PopSetPerfLevels(). The initialization can occur as before, when it was initializing PROCESSOR_PERF_LEVEL instead.

```
NTSTATUS
PopSetPerfLevels(
        IN    PPROCESSOR_STATE_HANDLER2 ProcessHandler
        )
{
        KAFFINITY                Processors, CurrentAffinity;
        KIRQL                    OldIrql;
        PKPRCB                   Prcb;
        NTSTATUS                 Status = STATUS_SUCCESS;
        ULONG                    i;
        ULONG                    PerfStatesCount = 0;
        UCHAR                    Freq;
        UCHAR                    KneeThrottleIndex = 0;
        UCHAR                    ThermalThrottleIndex = 0;
        PPROCESSOR_PERF_STATE    PerfStates = NULL;
        PPROCESSOR_PERF_STATE    TempStates;
        PPROCESSOR_POWER_STATE   PState;

        //
        // The first step is to convert the data that was passed to us
        // PROCESSOR_PERF_LEVEL over to PROCESSOR_PERF_STATE.
        //
        if (ProcessorHandler->NumPerfStates) {

              //
              // Because we are using going to allocate the PerfStates
              // array first so that that we can work on it, then copy
              // it to each processor, we can do that allocation from
              // paged pool.
              //
```

```
PerfStatesCount = ProcessorHandler->NumPerfStates;
PerfStates = ExAllocatePoolWithTag(
      PagedPool,
      PerfStatesCount * sizeof(PROCESSOR_PERF_STATE),
      'sPoP'
      );
if (PerfStates == NULL) {

      //
      // We can handle this case. We will set the
      // status code to an appropriate failure code
      // and we will clean up the existing processor
      // states. The reason we do that is because
      // this function only gets called if the current
      // states are invalid, so keeping the current ones
      // would make no sense.
      //
      status = STATUS_INSUFFICIENT_RESOURCES;
      goto PopSetPerfLevelsSetNewStates;

}

//
// Initialize each of the PROCESSOR_PERF_STATE entries
//
for (i = 0; i < PerfStatesCount; i++) {

      PerfStates[i].PercentFrequency =
            ProcessorHandler->PerfLeve[i].PercentFrequency;
      PerfStates[i].Power =
            ProcessorHandler->PerfLevel[i].Power;

}

//
// Analyze the PerfStates to determine which entries are
// linear/non-linear
//
PopAnalyzePerfStates( PerfStates, PerfStatesCount );

//
// Calculate the increase level, decrease level, and
// increase/decrease time
//
CalculateIncreaseLevel( PerfStates, PerfStatesCount );
CalculateDecreaseLevel( PerfStates, PerfStatesCount );
CalculateIncreaseDecreaseTime(
      PerfStates,
      PerfStatesCount,
      PerfHandler
      );

//
// Calculate where the Knee in the performance curve is
//
for (i=PerfStatesCount; i >= 1; i++) {
```

```
                        if (PerfStates[i-1].Flags & POP_THROTTLE_NON_LINEAR){

                              KneeThrottleIndex = i-1;

                        }

                  }

            }

PopSetPerfLevelsSetNewStates:

            if (PerfStates) {

                  //
                  // We have perf states, so remember that in our
                  // capabilities
                  //
                  PopCapabilities.ProcessorThrottle = TRUE;

                  //
                  // Find the minimum throttle value >=
                  // PopIdleDefaultMinThrottle and the current maximum
                  // throttle value
                  //
                  PopCapabilities.ProcessorMinThrottle = POP_PERF_SCALE;
                  PopCapabilities.ProcessorMaxThrottle = 0;
                  for (i=0;i<ProcessorHandler->NumPerfStates;i++) {
                  Freq = PerfStates->PerfLevel[i].PercentFrequency;
                  if ((Freq < PopCapabilities.ProcessorMinThrottle) &&
                        (Freq >= PopIdleDefaultMinThrottle)) {

                        PopCapabilities.ProcessorMinThrottle = Freq;

                  }
                  if ((Freq > PopCapabilities.ProcessorMaxThrottle) &&
                        (Freq >= PopIdleDefaultMinThrottle)) {

                        PopCapabilities.ProcessorMaxThrottle = Freq;
                        ThermalThrottleIndex = i;

                  }
            }

            //
            // There better be SOME speed we can run at.
            //
            ASSERT(PopCapabilities.ProcessorMaxThrottle >=
                  PopIdleDefaultMinThrottle);

      } else {

            //
            // We don't have any perf sates, so remember that in our
            // capabilities
            //
            PopCapabilities.ProcessorThrottle = FALSE;
```

```
        PopCapabilities.ProcessorMaxThrottle = POP_PERF_SCALE;
        PopCapabilities.ProcessorMinThrottle = POP_PERF_SCALE;
}

//
// Initialize the PPROCESSOR_POWER_STATE for each processor
//
Processors = KeActiveProcessors;
CurrentAffinity = 1;
while (Processors) {

        if (!(Processors & CurrentAffinity) {

                CurrentAffinity <<= 1;

        }

        //
        // Remember that we did this processor and make sure that
        // we are actually running on that processor. This ensures
        // that we are synchronized with the DPC and idle loop
        // routines
        //
        Processors &= ~CurrentAffinity;
        KeSetSystemAffinityThread(CurrentAffinity);
        CurrentAffinity <<= 1;

        //
        // To make sure that we aren't pre-empted, we must raise
        // to DISPATCH_LEVEL
        //
        KeRaiseIrql(DISPATCH_LEVEL, &OldIrql );

        //
        // Get the PRCB and PPROCESSOR_POWER_STATE structures that
        // we will need to manipulate
        //
        Prcb = KeGetCurrentPrcb();
        PState = &Prcb->PowerState;

        //
        // Remember what our thermal limit is
        //
        PState->ThermalThrottleLimit =
              PopCapabilities.ProcessorMaxThrottle;
        PState->ThermalThrottleIndex = ThermalThrottleIndex;

        //
        // To get the bookkeeping to work out correctly, we will
        // set the throttle to 0% (which is not possible), set the
        // current index to the last state, and set current tick
        // count to the current time.
        //
        PState->CurrentThrottle = 0;
        PState->PerfTickCount = Time;
        If (PerfStatesCount) {
```

```
            PState->CurrentThrottleIndex = PerfStatesCount - 1;

        } else {

            PState->CurrentThrottleIndex = 0;

        }

        //
        // Reset the Knee Index. This indicates where the knee
        // in the performance curve is
        //
        PState->KneeThrottleIndex = KneeThrottleIndex;

        //
        // Reset the Throttle Limit Index
        //
        PState->ThrottleLimitIndex = KneeThrottleIndex;

        //
        // If there are already perf states present for this
        // processor, then free them
        //
        if (PState->PerfStates) {

            ExFreeToNPagedLookasideList(
                  &PopProcPerfStateLookAsideList,
                  PState->PerfStates
                  );
            PState->PerfStates = NULL;
            PState->PerfStatesCount = 0;

        }

        //
        // At this point, we have to distinguish our behavior based
        // on whether or not we have perf states
        //
        if (PerfStates) {

            //
            // We do, so let allocate some memory and make a copy
            // of the template that we already created.
            //
            TempStates = ExAllocateFromNPagedLookasideList(
                  &PopProcPerfStateLookAsideList
                  );
            if (TempStates == NULL) {

                  //
                  // Not being able to allocate this structure
                  // is surely fatal. The only way to get around
                  // it (I think) is to break out of this case
                  // and treat it as if there are no PerfStates
                  // available.
                  //
                  statuts = STATUS_INSUFFICIENT_RESOURCES;
```

```
                        KeBugCheckEx(
                              INTERNAL_POWER_FAILURE,
                              6,
                              STATUS_INSUFFICIENT_RESOURCES,
                              __LINE__,
                              0
                              );

                  } else {

                        //
                        // Copy the template to the one associated with
                        // the processor
                        //
                        RtlCopyMemory(
                              TempStates,
                              PerfStates,
                              PerfStatesCount *
                                    sizeof(PROCESSOR_PERF_STATES)
                              );
                        PState->PerfStates = TempStates;
                        PState->PerfStatesCount = PerfStatesCount;

                  }

            }

            //
            // Update the processor throttle function
            //
            if (PState->PerfStates) {

                PState->PopSetThrottle =
                      ProcessorHandler->SetPerfLevel;

            } else {

                  PState->PopSetThrottle = NULL;

            }

            //
            // Update the processor throttle (since we are already
            // running on the target processor
            //
            PopUpdateProcessorThrottle();

            //
            // We can now return to our previous IRQL
            //
            KeLowerIrql( OldIrql );
      }

      //
      // Return to the proper affinity
      //
      KeRevertToUserAffinityThread();
```

```
        //
        // Return whatever status code we have
        //
        return status;
}
```

This function has been extensively changed from the existing code base. The first and most obvious change is the fact that we allocate per-processor perf state arrays. This means that we have to be able to handle the case where we cannot allocate such an array. The solution to this problem is to basically treat that failure as an inability of the system to throttle.

The changes to this function also means that a great deal of additional intelligence will have to be present in `PopUpdateProcessorThrottle()`.

The purpose of this algorithm is to flush each processor from using the old processor state tables. This is accomplished by running the thread in the context of each processors. Since this structure is only accessed within the context of a `TimerDPC` and the `IdleThread`, which both run at DPC level, and thus cannot be pre-empted, this guarantees that neither the `IdleThread` nor the `TimerDPC` can be running. If neither are running, then neither are using the old copy of the structure, which means that it can be safely freed and the new one used in its place.

The only potential problem with this algorithm is dealing with a potential low memory situation. It is not acceptable to share copies of the `PerfStates` structure among multiple processors. That leaves as solutions either guaranteeing that we don't run into low memory problem or issuing a bugcheck if we do. To avoid running into low memory problems, the `PerfStates` structures can be allocated from an `NPAGED_LOOKASIDE` list. A further optimization is that if the old `PerfStates` structure is the same size or larger than the new one, it could be used instead.

5.4 PopUpdateAllThrottles()

This function is used to update all the throttles simultaneously.

```
VOID
PopUpdateAllThrottles(
        VOID
        )
{
        KAFFINITY    Processors;
        KAFFINITY    CurrentAffinity;
        KIRQL        OldIrql;

        Processors = KeActiveProcessors;
        CurrentAffinity = 1;
        while (Processors) {

                if (Processors & CurrentAffinity) {
```

```
                        KeSetSystemAffinityThread(CurrentAffinity);

                        //
                        // We must call PopUpdateProcessorThrottle
                        // at DISPATCH_LEVEL
                        //
                        KeRaiseIrql( DISPATCH_LEVEL, &OldIrql );
                        PopUpdateProcessorThrottle();
                        KeLowerIrql( OldIrql );

                }
                CurrentAffinity <<= 1;

        }
        KeRevertToUserAffinityThread();
}
```

The principle change to this code is that we no longer have an early exit case if there are no perf states registered. In theory, we could add a check that would look at `PopCapabilities.ProcessorThrottle`.

The other change is that we will always call `PopUpdateProcessorThrottle` at `DISPATCH_LEVEL`. This will guarantee that the DPC routine will not be able to pre-empt the routine.

5.5 PopUpdateProcessorThrottles()

```
VOID
PopUpdateProcessorThrottles(
        VOID
        )
{
        PKRPRCB                     Prcb;
        PPROCESSOR_PERF_STATE       PerfStates;
        PPROCESSOR_POWER_STATE      PState;
        UCHAR                       I;
        UCHAR                       Index;
        UCHAR                       NewLimit;
        UCHAR                       PerfStatesCount;
        ULONG                       IdleTime
        ULONG                       Time;

        //
        // Get the PowerState structure from the PRCB
        //
        Prcb = KeGetCurrentPrcb();
        PState = &Prcb->PowerState;

        //
        // Make sure that this processor supports throttling
        //
        If (PState->PopSetThrottle == NULL) {
```

```
            Return;

        }

        //
        // Get the current information such as curren throttle, current
        // throttle index, current system time, current idle time
        //
        NewLimit = PState->CurrentThrottle;
        Index = PState->CurrentThrottleIndex;
        Time = CUR_TIME(Prcb);
        IdleTime = Prcb->IdleThread->KernelTime;

        //
        // We will need to refer to these frequently
        //
        PerfStates = PState->PerfStates;
        PerfStatesCount = PState->PerfStatesCount;

        //
        // If we are on AC, then we always want to run at the highest
        // possible speed. Also the same algorithm is used on DC if the
        // dynamic throttling policy is PO_THROTTLE_NONE.
        //
        if ((PopPolicy == &PopAcPolicy) ||
            (PopPolicy->DynamicThrottle == PO_THROTTLE_NONE)) {

            //
            // We precompute what the max throttle should be
            //
            Index = PState->ThermalThrottleIndex;
            NewLimit = PerfStates[Index].PercentFrequency;

        } else {

            //
            // We are on DC, apply the appropriate heuristics based on
            // the dynamic throttling policy.
            //
            switch (PopPolicy->DynamicThrottle) {
            case PO_THROTTLE_CONSTANT:

                //
                // We have pre-computed the optimal point on the
                // already. So, we might as well use that.
                //
                Index = PState->KneeThrottleIndex;
                NewLimit = PerfStates[Index].PercentFrequency;

                //
                // Set the constant flag and clear the degraded flag
                //
                PState->Flags &= ~PSTATE_DEGRADED_THROTTLE;
                PState->Flags |= PSTATE_CONSTANT_THROTTLE;
                PState->Flags |= PSTATE_ADAPTIVE_THROTTLE;
                break;
```

```
case PO_THROTTLE_DEGRADE:

      //
      // We calculate the limit of the degrade throttle
      // on the fly.
      //
      Index = PState->ThrottleLimitIndex;
      NewLimit = PerfStates[Index].PercentFrequency;

      //
      // Set the degraded flag and clear the constant flag
      //
      PState->Flags &= ~PSTATE_CONSTANT_THROTTLE;
      PState->Flags |= PSTATE_CONSTANT_THROTTLE;
      PState->Flags |= PSTATE_ADAPTIVE_THROTTLE;
      break;

case PO_THROTTLE_ADAPTIVE:

      PState->Flags |= PSTATE_ADAPTIVE_THROTTLE;
      break;

default:
      // not implemented
      PoPrint(
            PO_THROTTLE,
            ("PopUpdateProcessorThrottle - unimplemented"
            " dynamic throttle %d\n",
            PopPolicy->DynamicThrottle)
            );
      break;
   }
}

//
// Check if we are over the thermal limit.
//
ASSERT(PState->ThermalThrottleLimit >=
      PopCapabilities.ProcessorMinThrottle);
if (NewLimit > PState->ThermalThrottleLimit) {

      PoPrint(
            PO_THERM,
            ("PopUpdateProcessorThrottle - new throttle limit %d"
            " over thermal limit %d\n",
            NewLimit,
            PState->ThermalThrottleLimit)
            );
      NewLimit = PState->ThermalThrottleLimit;
      Index = PState->ThermalThrottleIndex;

}

//
// Apply new throttle if it has changed.
//
if (NewLimit != PState->CurrentThrottle) {
```

```
                PoPrint(
                      PO_THROTTLE,
                      ("PopUpdateProcessorThrottle - Setting CPU throttle "
                      "to %d\n",NewLimit)
                      );
                PopSetThrottle(
                      PState,
                      PerfState,
                      Index,
                      Time,
                      IdleTime
                      );

          }

}
```

It should be noted that this function can only be called within the context of the target processor. This function does not acquire any spinlocks because it is running at DISPATCH_LEVEL, thus preventing the Timer DPC and the Idle Thread from running on this processor.

5.5 PopApplyThermalThrottle()

```
VOID
PopApplyThermalThrottle(
      VOID
      )
{
      //
      // <Code which is not relevant to this spec here>
      //
#if DBG
      PoPrint(
            PO_THERM,
            ("Thermal - Zone %p - %s - Thermal throttle = %d.%dn",
            thermalZone, t,
            (thermalThrottle / 10),
            (thermalThrottle % 10))
            );
      PoPrint(
            PO_THERM,
            ("Thermal - Zone %p - %s - Forced throttle = %d.%d\n",
            thermalZone, t,
            (forcedThrottle / 10),
            (forcedThrottle % 10))
            );
#endif

      //
      // Set limit on effected processors
      //
      currentAffinity = 1;
      processors = KeActiveProcessors;
```

```
do {

      if (~(processors & currentAffinity) {

            currentAffinity <<= 1;
            continue;

      }

      processors &= ~currentAffinity;

      //
      // We must run on the target processor
      //
      KeSetSystemAffinityThread(currentAffinity);
      pState = &(KeGetCurrentPrcb()->PowerState);

      //
      // We need to be running at DISPATCH_LEVEL to access the
      // structures referenced within the pState…
      //
      KeRaiseIrql( DISPATCH_LEVEL, &OldIrql );

      //
      // Convert throttles to processor bucket size. We need to
      // do this in the context of the target processor processor
      // to make sure sure that we get correct set of perf levels
      //
      PopRoundThrottle(
            (UCHAR)(thermalThrottle/PO_TZ_THROTTLE_SCALE),
            &thermalLimit,
            NULL,
            &thermalLimitIndex,
            NULL
            );
      PopRoundThrottle(
            (UCHAR)(forcedThrottle/PO_TZ_THROTTLE_SCALE),
            &forcedLimit,
            NULL,
            &forcedLimitIndex,
            NULL
            );
#if DBG
      PoPrint(
            PO_THERM,
            ("Thermal - Zone %p - %d - Thermal Limit = %d\n",
            thermalZone, Prcb->Number, thermalLimit)
            );
      PoPrint(
            PO_THERM,
            ("Thermal - Zone %p - %d - Forced Limit = %d\n",
            thermalZone, Prcb->Number, forcedLimit)
            );
#endif

      //
```

```
// Figure out which one we are going to use
//
limit = (thermalProcessors & currentAffinity) ?
      thermalLimit : forcedLimit;
index = (thermalProcessors & currentAffinity) ?
      thermalLimitIndex : forcedLimitIndex;

//
// Next affinity mask
//
currentAffinity <<= 1;

//
// Does this processor support throttling?
//
if (pState->PopSetThrottle == NULL) {

      KeLowerIrql( OldIrql );
      continue;

}

//
// Check processors limit for a change
//
if (limit > PopCapabilities.ProcessorMaxThrottle) {

      PoPrint(
            PO_THERM,
            ("PopThrottle: Limit (%d) > Scale (%d)\n",
            limit,
            PopCapabilities.ProcessorMaxThrottle)
            );
      limit = PopCapabilities.ProcessorMaxThrottle;

} else if (limit < PopCapabilities.ProcessorMinThrottle) {

      PoPrint(
            PO_THERM,
            ("PopThrottle: Limit (%d) < MinThrottle "
            "(%d)\n",
            limit,
            PopCapabilities.ProcessorMinThrottle)
            );
      limit = PopCapabilities.ProcessorMinThrottle;

}

if (pState->ThermalThrottleLimit != limit) {

      pState->ThermalThrottleLimit = limit;
      pState->ThermalThrottleIndex = index;

}

//
// Revert back to our previous IRQL
```

```
            //
            KeLowerIrql( OldIrql );

        } while (processors);

        KeRevertToUserAffinityThread();

        //
        // Apply thermal throttles if necessary. Note we always do this
        // whether or not the limits were changed. This routine also gets
        // called whenever the system transitions from AC to DC, and that
        // may also require a throttle update due to dynamic throttling.
        //
        PopUpdateAllThrottles();
}
```

5.6 PopRoundThrottle()

```
VOID
PopRoundThrottle(
        IN UCHAR Throttle,
        OUT OPTIONAL PUCHAR RoundDown,
        OUT OPTIONAL PUCHAR RoundUp,
        OUT OPTIONAL PUCHAR RoundDownIndex,
        OUT OPTIONAL PUCHAR RoundUpIndex
        )
{
KIRQL                   OldIrql;
PKPRCB                  Prcb;
PPROCESSOR_PERF_STATE   PerfStates;
PPROCESSOR_POWER_STATE  Pstate;
UCHAR                   Low;
UCHAR                   LowIndex;
UCHAR                   High;
UCHAR                   HighIndex;
ULONG                   I;

//
// We need to get this processor's power capabilities
//
Prcb = KeGetCurrentPrcb();
PState = &(Prcb->PowerState);

//
// Make sure that we are synchronized with the Idle thread
// and other routines that access these data structures
//
KeRaiseIrql( DISPATCH_LEVEL, &OldIrql );
PerfStates = PState->PerfStates;

//
// Does this processor support throttling?
//
if (PState->PopSetThrottle == NULL) {
```

```
        if (ARGUMENT_PRESENT(RoundUp)) {

            *RoundUp = Throttle;
            if (ARGUMENT_PRESENT(RoundUpIndex)) {

                *RoundUpIndex = 0;

            }

        }
        if (ARGUMENT_PRESENT(RoundDown)) {

            *RoundDown = Throttle;
            if (ARGUMENT_PRESENT(RoundDownIndex)) {

                *RoundDownIndex = 0;

            }

        }
        KeLowerIrql( OldIrql );
        return;

    }

    //
    // Check if the supplied throttle is out of range
    //
    if (Throttle <= PopCapabilities.ProcessorMinThrottle) {

        Throttle = PopCapabilities.ProcessorMinThrottle;

    } else if (Throttle >= PopCapabilities.ProcessorMaxThrottle) {

        Throttle = PopCapabilities.ProcessorMaxThrottle;

    }

    //
    // Initialize our search space to something reasonable
    //
    Low = High = PerfStates[0].PercentFrequency;
    LowIndex = HighIndex = 0;

    //
    // Look at all the available perf states
    //
    for ( I = 0; I < PopPerfLevelCount; I++) {

        if ((Low > Throttle) &&
            (PerfStates[I].PercentFrequency < Low)) {

            Low = PerfStates[I].PercentFrequency;
            LowIndex = I;

        } else if (PerfStates[I].PercentFrequency > Low) {
```

```
                Low = PerfStates[I];
                LowIndex = I;

            }

            if ((High < Throttle) &&
                (PerfStates[I].PercentFrequency > High)) {

                High = PerfStates[I];
                HighIndex = I;

            } else if (PerfStates[I].PercentFrequency < High) {

                High = &PerfStates[I];
                HighIndex = I;

            }

        }

        //
        // Revert back to our previous IRQL.
        //
        KeLowerIrql( OldIrql );

        //
        // Fill in the pointers provided by the caller
        //
        if (ARGUMENT_PRESENT(RoundUp)) {

            *RoundUp = High;
            if (ARGUMENT_PRESENT(RoundUpIndex)) {

                *RoundUpIndex = HighIndex;

            }

        }

        if (ARGUMENT_PRESENT(RoundDown)) {

            *RoundDown = Low;
            if (ARGUMENT_PRESENT(RoundDownIndex)) {

                *RoundDownIndex = LowIndex;

            }
        }
    }
```

The changes in this routine include an optimization for dealing with the case where the desired throttle is above/below the maximum/minimum. It also now returns the index into the `PerfStates` array that correspond with the rounded up and rounded down values.

5.7 PopCompositeBatteryDeviceHandler()

This routine is the one that is notified when the total battery remaining level changes. For adaptive throttling to work, whenever a new battery notification comes in, we need to update the current ThrottleLimitIndex.

```
VOID
PopCompositeBatteryDeviceHandler(
      IN PDEVICE_OBJECT DeviceObject,
      IN PIRP           Irp,
      IN PVOID          Context
      )
{
      <...>

      if (NT_SUCCESS(Irp->IoStatus.Status) {

            //
            // Handle the completed request
            //
            switch (PopCB.State) {
            <...>
            case PO_CB_READ_STATUS:
                  <...>
                  if (Policy == &PopDCPolicy) {

                        <...>

//
// This is kind of silly, but since we
// want to minimize our synchronization
// elsewhere, we have to examine every
// processor's PowerState and update the
// ThrottleLimitIndex on each. This
// may eventually be the smart thing to
// do if not all processors support the
// same set of states.
//
currentAffinity = 1;
processors = KeActiveProcessors;
while (processors) {

      if (!(processors & currentAffinity)) {

            currentAffinity <<= 1;
            continue;

      }

      KeSetSystemThreadAffinity(
            currentAffinity
            );
      currentAffinity <<= 1;

      //
      // We need to run at DISPATCH_LEVEL to
```

```
// properly synchronize access to these
// power structures
//
KeRaiseIrql( DISPATCH_LEVEL, &OldIrql );

Prcb = KeGetCurrentPrcb();
PState = &(Prcb->PState);
PerfStates = PState->PerfStates;
PerfStatesCount =
      PState->PerfStatesCount;
For (I = PState->KneeThrottlIndex;
      I < PerfStatesCount;
      I++) {

      If (PerfState[I].MinCapacity >=
            PopCB.Status.Capacity) {

            Break;

      }

}
PState->ThrottleLimitIdnex = I;

//
// We can revert back to our previous
// IRQL now
//
                  KeLowerIrql( OldIrql );

                  }

                  KeRevertToUserAffinityThread();

                  <...>

            <...>

            }

      <...>

      }

<...>

      }
}
```

EXHIBIT C

1.0 Document Overview

1.1 Document Purpose

The purpose of this paper is to describe a possible implementations for an Adaptive Throttling Policy. The intent of this document is to gather a permanent record of the design and thought process for patent and implementation verification purposes

1.2 Revision History

- V0.1, September 3rd, 2000. Initial Revision
- V0.2, September 6th, 2000. Per-Processor Performance State information
- V0.3, September 11th, 2000. Review comments merged
- V0.4, September 18th, 2000. Thermal Integration
- V0.9, September 19th, 2000. Battery Integration. Document is Complete.
- V0.91, September 25th, 2000. Typos & Corrections
- V1.00, October 29th, 2000. First implementation changes

2.0 Design Vision

2.1 What is Adaptive Throttling?

When a computer is running on batteries, it is not desirable to always run the CPU at its maximum available frequency. For example, if the computer is idle, because the user is reading a Microsoft Word document, running the CPU at full frequency merely drains the battery much more quickly.

Adaptive Throttling is the idea that the CPU should run at the maximum frequency required to fulfill the user's current needs. For example, while the user is reading the Microsoft Word document, the CPU should be throttled to its lowest possible frequency to save power. As soon as the user hits the page-down key or does anything else that requires the CPU, the CPU should be throttled back up to the frequency that places the CPU closest to being 100% busy as possible.

In practice, the system should only pick the highest-throttle for each of the voltage states supported. The reason being that if the CPU is idle enough of the time, it will spend a large portion of time in the C2 state, which effectively means that it has been throttled to the correct level. Since Microsoft has invested a great of energy into getting the C-State algorithms correct, this implementation should leverage that.

In the code, this is referred to as `PO_THROTTLE_ADAPTIVE`.

2.2 What is Degraded Throttling?

Degraded Throttling is a subset of Adaptive Throttling. The difference is that Adaptive Throttling does not put a cap on the maximum frequency that can be selected whereas Degraded Throttling does. This is useful in enforcing a policy where the user is willing to

trade away some performance for longer battery life. It is particularly useful in situations where the CPU is stuck "busy-waiting". Typically the Operating System should begin by placing the cap at the lower-voltage, highest-throttle state and decreasing to lower throttle states as battery capacity diminishes.

As an example, when the user hits the page-down key, the CPU might revert to 50% of its maximum frequency while the next part of the document is read from the disk and the screen is re-drawn. This might take a little longer than if the CPU had reverted to 100%, but it assumed that there would some saving in run the processor at a lower frequency for a longer period of time.

In the code, this is referred to as `PO_THROTTLE_DEGRADE`.

2.3 What is Constant Throttling?

Constant throttling is a subset of Adaptive Throttling and is very similar to Degraded Throttling. They both start at the lowest-voltage highest-frequency state, but unlike Degraded Throttling, Constant Throttling will never **force** the maximum throttle to goto a lower frequency state. The throttle is actually allowed to go to a lower frequency if so desired, but not forced to do so.

In the code, this is referred to as a `PO_THROTTLE_CONSTANT`.

2.4 Why Implement Adaptive Throttling?

We currently handle a few scenerios badly. And OEM perception, with the help of AMD and Intel, is that there are other scenarios that we handle badly which we think we handle well, which leads to them shipping their own drivers and crapplets.

- Machine is mostly or completely idle: We currently handle this well. We put the CPU into C3 via the SLP# signal. The CPU is as deeply asleep as it would be in S1.
- Machine is used to play Ms. PacMan, or any other app that eats some but not all of the CPU bandwidth. We currently handle this poorly. We put the CPU into C1 whenever we hit the idle loop, meaning that the CPU consumes quite a bit of power. Part of what makes this scenario tricky is that we don't know if the app will gracefully degrade if we take away CPU bandwidth. We should handle this scenario at first by trying to match CPU performance with CPU bandwidth. We can make the CPU bandwidth degrade over time if the Degrade policy is chosen.
- Machine is running with apps consuming all CPU. We handle this poorly. The CPU stays in C0. Out attitude in the past has been to find these apps and get them to fix it. Unfortunately, this attitude hasn't paid off.
- Machine is being used to play a DVD. We handle this poorly. The CPU stays in C0. The scenario is worthy of mention because it has special attributes. First, we know that restricting the CPU bandwidth will result in degraded performance, but the app will still run. Second, we could potentially find out how long the DVD is, allowing us to know how much total performance is needed. Third, on the laptops that we have looked at, DVD playback tends not to use all available CPU bandwidth.

2.5 Hardware Issues

Some concerns regarding the current and future generation of processors are important considerations.

- The hardware latency for changing the voltage is potentially long enough that it can cause CPU-availability problems. For example, a soft-modem can drop a connection if it isn't serviced every 10ms or that it will never make a connection if it isn't serviced every 2ms during the "training phase" (at the beginning of the call). Intel's best-case voltage switching time is around 2ms. Dell machines seem to take 24 retries, which brings them into the > 50ms range. AMD is currently at 200us, with an occasional retry. Transmeta claims to be at 20us
- The hardware latency for changing the CPU frequency without changing the voltage is around 3us. Unfortunately, we don't have a way of telling the kernel that the latency for entering a state is large if your're coming from a different voltage but tiny if you're staying with the same voltage. We could make this assumption directly in the kernel.
- CPU frequency is adjusted by causing the chipset to deassert the CLK_RUN# signal N out of M cycles, where M is usually 8. Deasserting CLK_RUN# is also what happens when we hit the C2 idle state
- Using the C3 idle state consumes significantly less power than the C2 power state. Throttling while using C3 causes the CPU to run longer, effectively putting it in C2 for part of the time that it could be in C3. This means that throttling while using C3 wastes power. IBM and others have shown us empirical data to support this.

2.6 Integration Issues

There are several minor issues that must be handled for the system to perform optimally.

- The system should return to highest-frequency highest-voltage when beginning any sort of power management operation (Sleep or Hibernate). This is essential during Hibernate to ensure that writing the hibernate file does not become a CPU-bound operation. It also insures that if the machine is transitioning to a Sleep or Hibernate state due to battery considerations, that the machine spends the minimal amount of time to make the transition. Just before entering the Sleep state, the processors should be returned to the lowest-voltage state possible.
- The implementation should respect the result of the thermal policy manager. If the thermal policy forces the throttle to be reduced, the Adaptive Throttling manager should not increase the throttle past that point
- The Operating System will perform better if we return to the lowest-voltage highest frequency state during any period of heavy C3 activity.

2.7 Time Management

For ease of integration with the existing Idle Promotion code base, all time units will be kept track of in terms of TickCounts. These are the units used in `Prcb->KernelTime`, `Prcb->UserTime`, and `Thread->KernelTime`. The following define will be used to establish the current system time:

```
#define POP_CUR_TIME(X)   (X->KernelTime + X->UserTime)
```

Where X represents a pointer to the current PRCB. The reason that we take in a pointer to the PRCB is that it is more efficient to get the PRCB once and then kept track of it in a local variable, even though KeGetCurrentPrcb() is expanded into an in-line call.

3.0 Data Structures

3.1 PROCESSOR_PERF_STATE

This data structured is defined in ntos\pop.h. This structure replaces (in the kernel only) PROCESSOR_PERF_LEVEL.

```
typedef struct {
    UCHAR     PercentFrequency;    // max = 100
    UCHAR     MinCapacity;         // Percentage
    USHORT    Power;               // milliwatts
    UCHAR     IncreaseLevel;       // goto higher freq
    UCHAR     DecreaseLevel;       // goto lower freq
    USHORT    Flags;               // Used for Flags
    ULONG     IncreaseTime;        // goto higher freq
    ULONG     DecreaseTime;        // goto lower freq
    ULONG     IncreaseCount;       // goto higher freq
    ULONG     DecreaseCount;       // goto lower freq
    ULONGLONG PerformanceTime;     // for tick count
} PROCESSOR_PERF_STATE, *PPROCESSOR_PERF_STATE
```

The kernel will allocate an array of these structures and store the pointer to the array in the Processor's PRCB.

The following Flags are defined as being available:

```
#define POP_THROTTLE_NON_LINEAR          0x1
```

PercentFrequency is the normalized representation of frequency that this performance state represents. The highest performance state has a frequency of 100%, if it is available. This avoids the problem of dealing with faster and faster CPUs. Under this mechanism, a CPU that has a max speed of 450MHz uses the same algorithm as one that runs at 700Mhz.

MinCapacity is used to represent the minimum battery remaining capacity that is required for the CPU to be in this state. It should be noted that this only applies if the machine is running on DC since the relevant throttling policies are only available then. This value is expressed as a percentage.

IncreaseLevel and DecreaseLevel are the boundaries of the bucket that defines the current state. If the CPU is busier than IncreaseLevel, the Operating System should pick a higher processor frequency. If the CPU is less busy than DecreaseLevel, the OperatingSystem should pick a lower processor frequency.

It should be noted that `PercentFrequency` will be higher than `IncreaseLevel` since the only way to reach `PercentFrequency` is for the Operating System to be 100% busy for the given frequency. That is, the CPU must be using every single cycle allocated to it in order to be running at `PercentFrequency` level of business. In order to allow for promotion in cases where the system is not quite at that level of business, `IncreaseLevel` must be a smaller value than `PercentFrequency`. If `IncreaseLevel` is higher than `PercentFrequency`, then promotion can never occur.

`IncreaseCount` and `DecreaseCount` are used to keep track of the number of transitions **from** this state to another performance state. Transitions to a lower performance state cause an increase in `IncreaseCount`. Transitions to a higher performance state cause an increase in `DecreaseCount`.

`PerformanceTime` is used to keep trace of the number of ticks that are spent at this performance level. This value is updated whenever the processor switches to a different performance state. When the user queries the performance information, the current elapsed time at this state is added to `PerformanceTime` for the current performance state.

3.2 PROCESSOR_POWER_STATE

This structure is defined in sdkinc\ntpoaepi.h. This structure already exists but must be grown to accommodate more information. The structure was removed from the published header file and moved to the private ntos\inc\procpowr.h header file. Changed elements are listed in red.

```
typedef struct {
    PPROCESSOR_IDLE_FUNCTION IdleFunction;
    ULONG                    Idle0KernelTimeLimit;
    ULONG                    Idle0LastTime;
    PVOID                    IdleState;
    ULONGLONG                LastCheck;
    PROCESSOR_IDLE_TIMES     IdleTimes;
    ULONG                    IdleTime1;
    ULONG                    PromotionCheck;
    ULONG                    IdleTime2;
    UCHAR                    CurrentThrottle;
    UCHAR                    ThermalThrottleLimit;
    UCHAR                    Spare1[2];
    UCHAR                    ThermalThrottleIndex;
    UCHAR                    CurrentThrottleIndex;
    ULONG                    Spare2[2];
    ULONG                    PerfSystemTime;
    ULONG                    PerfIdleTime;
    // Temp for debugging…
    ULONGLONG                DebugDelta;
    ULONG                    DebugCount;
```

```
    ULONG                           LastSysTime;
    ULONGLONG                       TotalIdleStateTime[3];
    ULONG                           TotalIdleTransitions[3];
    ULONG                           Spare3;
    ULONGLONG                       PreviousC3StateTime;
    UCHAR                           KneeThrottleIndex;
    UCHAR                           ThrottleLimitIndex;
    UCHAR                           PerfStatesCount;
    UCHAR                           ProcessorMinThrottle;
    UCHAR                           ProcessorMaxThrottle;
    UCHAR                           LastBusyPercentage;
    UCHAR                           LastC3Percentage;
    UCHAR
        LastAdjustedBusyPercentage;
    UCHAR                           Spare1[1];
    ULONG                           Flags;
    ULONG                           PromotionCount;
    ULONG                           DemotionCount;
    LARGE_INTEGER                   PerfCounterFrequency;
    ULONG                           PerfTickCount;
    KTIMER                          PerfTimer;
    KDPC                            PerfDpc;
    PPROCESSOR_PERF_STATE           PerfStates;
    PSET_PROCESSOR_THROTTLE         PerfSetThrottle;
} PROCESSOR_POWER_STATE, *PPROCESSOR_POWER_STATE;
```

The `PerfDpc` and `PerfTimer` variables are convenient storage areas for the context information that will be required to fire a periodic timer to make sure that the CPU is not too busy for its current performance level. The `Flags` field will be used to store useful state information. The only useful defines (as of this document) are:

```
#define PSTATE_SUPPORTS_THROTTLE    0x1
#define PSTATE_ADAPTIVE_THROTTLE    0x2
#define PSTATE_DEGRADED_THROTTLE    0x4
#define PSTATE_CONSTANT_THROTTLE    0x8
```

It should be noted that `PSTATE_ADAPTIVE_THROTTLE` is what turns on the entire throttling behavior and that the other flags are used to modify the behavior. `PSTATE_DEGRADED_THROTTLE` and `PSTATE_CONSTANT_THROTTLE` are also mutually exclusive flags.

The `PerfSystemTime` and `PerfIdleTime` structure are used to keep track of previous values to calculate the important time deltas. `PerfSystemTime` is used to store the amount of system time previously elapsed, as expressed by CUR_TIME(`Prcb`). `PerfIdleTime` is used to store the amount of time that was spent in the idle thread as expressed by `Thread->KernelTime`.

PerfStates is a pointer to the PROCESSOR_PERF_LEVEL array associated with this processor. Each processor must have unique copy of this structure to maintain per-processor information about the amount of time spent in each state.
PerfStatesCount is the number of elements within this array.

CurrentThrottleIndex is the index in the PerfStates array that has currently been selected. This has been done using a for loop to find the entry in the PerfStates array that corresponds to CurrentThrottle.

KneeThrottleIndex is the index in the PerfStates array that represents the lowest-voltage highest frequency part of the curve. This value is pre-calculated since the Degraded and Constant Throttling policies depend upon it.

ThrottleLimitIndex is the index in the PerfStates array that represents the maximum state that is acceptable under the current policy. The maximum of this value is the KneeThrottleIndex. This value is modified by the Kernel's Battery subsystem whenever it receives new battery capacity remaining information.

ThermalThrottleIndex is the index in the PerfStates array that represents the maximum state that is acceptable based upon the thermal throttle. This value is used under all policies.

ProcessorMinThrottle and ProcessorMaxThrottle are the minimum and maximum frequencies available for this processor. This information used to be stored system-wide in the PopCapabilities, but has been moved to per-processor. In theory, it is possible to implement different minimum and maximum for each processors.

LastBusyPercentage is used to store the last value calculated by PopCalculateBusyPercentage() for debugging purposes.
LastC3Percentage is used to store the last value calculated by PopCalculateC3Percentage() for debugging purposes.
LastAdjustedBusyPercentage is used to store the modified busy frequency after all the extraneous factors have been calculated. It is useful for providing debug information.

PromotionCount keeps track of all the promotions done for this processor.
DemotionCount keeps track of all the demotions done for this processor. This allows for user applications to quickly see the number of transitions that have been made.

PreviousC3StateTime was the amount of time spent at C3 during the last successful throttle check. We need this time to compute the delta amount of time spent at C3 during the previous interval and thus determine the percentage time we spent at C3.

PerfTickCount is the CUR_TIME(Prcb) when the processor switches to a new performance state. When the processor switches to a new performance state, the

`CUR_TIME(Prcb)` minus `PerfTickCount` is added to the `PerformanceTime` bucket for the previous performance level.

`PerfCounterFrequency` is the stored value of `KeQueryPerformanceCounter()` to obtain the frequency rating of the counters. This is required since we want to minimize the number of calls we make and it's a good idea to cache this number. `PerfCounterFrequency` is used to make the transitions between values which are stored in PerformanceCounter units and those that are stored in Tick Counts.

`PerfSetThrottle` is the function that will get called when the Operating System wants to set a new throttle. The reason that this is in this structure instead of a global variable is that we want to properly synchronize access to it.

4.0 Algorithms

4.1 Setting a Throttle Level

The following code will be used to set a particular processor to a specific thermal level

```
VOID
FASTCALL
PopSetThrottle(
      PPROCESSOR_POWER_STATE  PState,
      PPROCESSOR_PERF_STATE   PerfStates,
      ULONG                   Index,
      ULONG                   SystemTime,
      ULONG                   IdleTime
      )
{
      NTSTATUS     Status;
      PKPRCB       Prcb;
      PKTHREAD     Thread;
      UCHAR        Current = PState->CurrentThrottleIndex;

      PoPrint(
            PO_THROTTLE,
            "PopSetThrottle: Index=%d (%d%%) at %ld (system)"
            " %ld (idle)\n",
            Index,
            PerfStates[Index].PercentFrequency,
            SystemTime,
            IdleTime
            )
            );

      //
      // Actually set the processor to the new throttle level
      //
      Status = PState->PerfSetThrottle(
            PerfStates[Index].PercentFrequency
            );
      if (!NT_SUCCESS(Status)) {
```

```
        //
        // If it didn't succeed, then don't update the
        // stats.
        //
        return;

    }

    //
    // Get the PRCB so that we can update the kernel and idle threads
    // The reason we do this is that a transition is usually non-zero
    // work, so we don't want the transitions to affect the free/busy
    // calcalutations.
    //
    Prcb = KeGetCurrentPrcb();
    Thread = Prcb->IdleThread;
    SystemTime = POP_CUR_TIME(Prcb);
    IdleTime = Thread->KernelTime;
    PoPrint(
          PO_THROTTLE,
          "PopSetThrottle: Index=%d (%d%%) now at %ld (system)"
          " %ld (idle)\n",
          Index,
          PerfStates[Index].PercentFrequency,
          SystemTime,
          IdleTime
          )
          );

    //
          // Update the booking for the current state
          //
          PerfStates[Current].PerformanceTime += (SystemTime -
                PState->PerfTickCount);

          //
          // Update the current throttle information
          //
          PState->CurrentThrottle = PerfStates[Index].PercentFrequency;
          PState->CurrentThrottleIndex = Index;

          //
          // Update our idea of what the current tick counts are
          //
          PState->PerfIdleTime = IdleTime;
          PState->PerfSystemTime = SystemTime;
          PState->PerfTickCount = SystemTimeCount;

          //
          // Remember how much we spent in C3 at this point
          //
          PState->PreviousC3StateTime = PState->TotalIdleState[2];
    }
```

This function, which will be called only on the target processor, while running either at DPC level or within the affinity of the target processor, will actually set the new throttle and update the bookkeeping.

4.2 Busy and C3 Detection

The following code can be called within the context of the target processor to determine how busy the CPU has been during the previous time period. This function would typically be called from the IdleThread, a DPC, or while running at DISPATCH_LEVEL.

```
UCHAR
PopCalculateBusyPercentage(
      PPROCESSOR_POWER_STATE  PState
      )
{
      PKPRCB      Prcb;
      PKTHREAD    Thread;
      UCHAR       Frequency;
      ULONGLONG   Idle;
      ULONG       Busy;
      ULONG       IdleTimeDelta;
      ULONG       CpuTimeDelta;

      Thread = KeGetCurrentThread();
      Prcb = CONTAINING_RECORD( PState, KPRCB, PowerState);

      IdleTimeDelta = Thread->KernelTime - PState->PerfIdleTime;
      CpuTimeDelta = CUR_TIME(Prcb) - PState->PerfSystemTime;
      Idle = (IdleTimeDelta * 100)/(CpuTimeDelta);

      //
      // We cannot be more than 100% idle, and if we are then we
      // are 0% busy (by definition), so apply the proper caps
      //
      if (Idle > 100) {

            Return 0;

      }
      Busy = 100 - Idle;
      Frequency = (UCHAR) (Busy * PState->CurrentThrottle /
            POWER_PERF_SCALE);

      //
      // Remember how busy we were. This will make debugging so much
      // easier.
      //
      Prcb->PowerState.LastBusyPercentage = frequency;
      return Frequency;
}
```

The Idle and Busy values represent a percentage of what the CPU was doing during the last interval. To simplify the math later on, these numbers are normalized against the current throttle value.

For example:

- If the CPU was 50% busy at 50%Throttle, that really means that the CPU was 25% busy at 100% throttle.
- If the CPU was 100% busy at 25% Throttle, that really means that the CPU was 25% busy at 100% throttle.
- If the CPU was 10% busy at 10% Throttle, that really means that the CPU was 1% busy at 100% throttle.

Similarly, the formula for detecting how much time the CPU has spent in C3 during a known interval is:

```
UCHAR
PopCalculateC3Percentage(
      PPROCESSOR_POWER_STATE  PState
      )
{
      PKPRCB            Prcb;
      ULONGLONG         CpuTimeDelta;
      ULONGLONG         C3;
      LARGE_INTEGER     C3Delta;

      Prcb = CONTAINING_RECORD( PState, KPRCB, PowerState);

      //
      // Calculate the C3 time delta in terms of Nanosecs.
      // The formulas for conversion are taken from
      // PopConvertUsToPerfCount
      //
      C3Delta.QuadPart = PState->TotalIdleState[2] -
            PState->PreviousC3StateTime;
      C3Delta.QuadPart = (US2SEC * US2TIME * C3Delta.QuadPart) /
            PState->PerfCounterFrequency.QuadPart;

      //
      // Now, calculate the CpuTimeDelta in terms of
      // NanoSeconds
      //
      CpuTimeDelta = (CUR_TIME(PRCB) - PState->PerfSystemTime) *
            KeTimeIncrement;

      //
      // Figure out the ratio of the two, and cap it
      // at 100%
      //
      C3 = C3Delta.QuadPart * 100 / CpuTimeDelta;
      If (C3 > 100) {
            Return 100;
      }

      //
      // Remember what it was --- this will make debugging so much
      // easier
      //
```

```
        Prcb->PowerState.LastC3Percentage = (UCHAR) C3;
        return (UCHAR) C3;
}
```

4.3 Calculating IncreaseLevel

To calculate what the upper bound for any PROCESSOR_PERF_STATE, the following rules apply:

```
VOID
PopCalculateIncreaseLevel(
      PPROCESSOR_PERF_STATE   CpuState,
      ULONG                   CpuStateCount
      )
{
      ULONG I;
      ULONG DeltaPerf;

      //
      // Optimization for case where there are no CpuStates
      //
      if (CpuStateCount == 0) {

            return;

      }

      //
      // This guarantees that we can never promote past this state
      //
      CpuState[0].IncreaseLevel = CpuState[0].PercentFrequency + 1;

      //
      // Calculate the increase Level
      //
      For (I=1; I < CpuStateCount;I++) {

            DeltaPerf = CpuState[I-1].PercentFreqency -
                  CpuState[I].PercentFreqency;
            DeltaPerf *= PopPerfIncreasePercentModifier;
            DeltaPerf /= POWER_PERF_SCALE;
            DeltaPerf += PopPerfIncreaseAbsoluteModifier;
            If (DeltaPerf > CpuState[I].PercentFrequency) {

                  DeltaPerf = POWER_PERF_SCALE + 1;

            } else {

                  DeltaPerf = CpuState[I].PercentFrequency -
                        DeltaPerf;

            }
            CpuState[I].IncreaseLevel = (UCHAR) DeltaPerf;
      }
}
```

It should be noted that the increase level will always result in the percentage business required for a promotion to the next higher throttle level, regardless of whether or not a voltage change is required. The reason that this is the case is because it is impossible to actually increase more than one level using the Idle Detection algorithm previously presented.

If its not desired that we should promote to a higher frequency within the same voltage band, than this could be accomplished by removing any non-linear states from the list of states or by forcing the increase level to be the same value used by the highest frequency state in the voltage range.

4.4 Calculating DecreaseLevel

To calculate what the lower bound for any PROCESSOR_PWER_STATE, the following rules apply:

```
VOID
PopCalculateDecreaseLevel(
      PPROCESSOR_PERF_STATE   CpuState,
      ULONG                   CpuStateCount
      )
{
      //
      // We will be required to walk the CpuState array several times
      // and the only way to safely keep track of which index we are
      // looking at versus the one we care about is to use to variable
// to keep track of indexes.
//
ULONG I, J;
ULONG DeltaPerf;

//
// Sanity Check
//
if (CpuStateCount == 0) {

      return;

}

//
// Set the decrease value for the last element in the array
//
CpuState[CpuStateCount-1].DecreaseLevel = 0;

//
// Calculate the decrease level
//
for (I = 0 ;I < (CpuStateCount - 1); I++) {

      if (I == (CpuStateCount-1)) {

            CpuState[I].DecreaseLevel = 0;
            Continue;
```

```
        +

        DeltaPerf = CpuState[I-1].PercentFrequency -
              CpuState[I+1].PercentFrequency;
        DeltaPerf *= PopPerfDecreasePercentModifier;
        DeltaPerf /= POWER_PERF_SCALE;
        DeltaPerf += PopPerfDecreaseAbsoluteModifier;

        if (DeltaPerf > CpuState[I+1].PercentFrequency) {

              DeltaPerf = 0;

        } else {

              DeltaPerf = CpuState[I+1].PercentFrequency -
              DeltaPerf;

        }
        CpuState[I].DecreaseLevel = (UCHAR) DeltaPerf;
}

//
// We want to eliminate demotions at the same voltage
// level, so guarantee that the decrease levels result
// in being set to the next voltage level...
//
I = 0;
while ( I < CpuStateCount ) {

        //
        // Find the next non-linear state. We assume that I
        // is currently pointing at the highest-frequency
        // state within a voltage band and we are interesting
        // in finding the highest-frequency state at the
        // next-lower voltage band.
        //
        for (J = I + 1; J < CpuStateCount; J++ ) {

              If (CpuState[J].->Flags & POP_THROTTLE_NON_LINEAR) {

                    Break;

              }

        }

        //
        // Want to find the previous state since that
        // will be the decrease limit that we will use
        //
        J--;

        //
        // Set the decrease limit to this new level
        //
        while (I < J) {
```

```
                CpuState[I]->.DecreaseLevel =
                      CpuState[J]->.DecreaseLevel;
                I++;

            }

            //
            // Skip the Jth state since it is the bottom of
            // the frequencies available for the current
            // voltage level. Note that we are skipping this
            // from I's point of view.
            //
            I++;

        }

}
```

This algorithm looks at the list of available states twice. The first time, it calculates the value that would be used to decrease the throttle to the next lower value. The second time, it calculate the value that would be used to decrease the throttle to the next lower non-linear state. The reason that this algorithm is used is because there are almost power savings to decreasing the frequency but keeping the voltage constant. Thus, any demotions should result in voltage changes.

The reason that decreasing the frequency but keeping the voltage produces few power savings is that with an aggressive C2 policy, running the CPU at a higher frequency while spending lots of time in C2 is equivalent to running the CPU at a lower frequency while spending no time in C2.

4.5 Calculting Increase/DecreaseTime

```
VOID
PopCalculateIncreaseDecreaseTime(
      PPROCESSOR_PERF_STATE          CpuState,
      ULONG                          CpuStateCount,
      PPROCESSOR_STATE_HANDLER2      PerfHandler
      )
{
      ULONG I;
      ULONG TickRate;
      ULONG Time;

      If (CpuStateCount == 0) {
            Return;
      }

      //
      // We can never increase from State 0
      //
      CpuState[0].IncreaseTime = (ULONG) -1;

      //
```

```
// Get the current tick rate
//
TickRate = KeQueryTimeIncrement();

//
// Loop over the next elements...
//
For (I = 1; I < CpuStateCount; I++ ) {

      //
      // Decrease Time of previous state
      // should be based on whether current state is
      // linear or not
      //
      CpuState[I-1].DecreaseTime =
            PerfHandler->HardwareLatency * 10 *
            PopPerfDecreaseTimeValue;
      If (CpuState[I].Flags & POP_THROTTLE_NON_LINEAR ||
            CpuState[I-1].Flags & POP_THROTTLE_NON_LINEAR)
      {
            CpuState[I-1].DecreaseTime *= 10;
      }
      Time += PopPerfDecreaseTimeValue;

      //
      // We do have some minimums that we must respect
      //
      If (Time < PopPerfDecreaseMinimumTime) {

            Time = PopPerfDecreaseMinimumTime;

      }

      //
      // Time is in Micro Seconds. Need to convert to tick
      // counts.
      //
      PerfStates[I-1].DecreaseTime = Time * US2TIME /
            TickRate + 1;

      //
      // Increase Time of current state should be
      // based on whether or not the state is
      // linear or not
      //
      CpuState[I].IncreaseTime =
            PerfHandler->HardwareLatency * 10 *
            PopPerfIncreaseTimeValue;
      If (CpuState[I-1].Flags & POP_THROTTLE_NON_LINEAR ||
            CpuState[I].Flags & POP_THROTTLE_NON_LINEAR)
      {
            CpuState[I].IncreaseTime *= 10;
      }
      Time += PopPerfIncreaseTimeValue;

      //
      // We do have some minimums that we must obey
```

```
            //
            If (Time < PopPerfIncreaseMinimumTime) {

                  Time = PopPerfIncreaseMinimumTime;

            }

            //
            // Time is in Micro Seconds. Need to conver to tick
            // counts.
            //
            PerfStates[I].IncreaseTime = Time * US2TIME /
                  TickRate + 1;
      }

      //
      // We can never decrease from the last state
      //
      I--;
      CpuState[I].DecreaseTime = (ULONG) -1;

}
```

4.6 Calculate MinCapacity

This routine is used to determine at what levels of battery capacity we start to force throtte when we are running on the degraded throttle.

```
VOID
PopCalculatePerfMinCapacity(
      PPROCESSOR_PERF_STATE   PerfStates,
      ULONG                   PerfStatesCount,
      PPROCESSOR_POWER_STATE  PState
)
{
      UCHAR I;
      UCHAR Knee = 0;
      UCHAR Num;
      UCHAR Total = PopPerfDegradeThrottleMinCapacity;
      UCHAR Width = 0;

      if (!PerfStatesCount) {

            return;

      }

      //
      // Calculate the Knee of the Curve. .. this is quick and avoids
      // having to pass a PRCB or other structure around
      //
      for (I = (UCHAR) PerfStatesCount; I >= 1; I--) {

            If (PerfStates[I-1].Flag & POP THROTTLE NON LINEAR) {

                  Knee = (I-1);
```

```
                Break;

        }

}

//
// Look at all the states that happen before the knee and
// set those to run only when the battery is fully charged
//
for (I = 0; I < PState->KneeThrottleIndex; I++) {

        //
        // Any of the steps before the knee are set to 100%
        //
        PerfStates[I].MinCapacity = 100;

}

//
// Calculate the range for which we clamp down the throttle
//
Num = PerfStatesCount - (PState->KneeThrottleIndex + 1);
if (Num != 0) {

        Width = Total / Num;

}

//
// Look at all the states from the Knee to the end. Starting
// at the highest value, set the min capacity and subtract the
// appropriate value to get the next min capacity.
//
for (I = PState->KneeThrottleIndex; I < PerfStatesCount; I++) {

        //
        // We put a floor onto how low we can force the throttle
        // down to. If this state is operating below that floor,
        // then we should set the MinCapacity to 0, which
        // reflects the fact that we don't want to degrade
        // past this point
        //
        if (PerfState[I].PercentFrequency <
                PopPerfDegradeThrottleMinFrequency) {

                //
                // We modify the min capacity for the
                // previous state since we don't ever
                // want to demote from that state.
                // Also, once we start being less than
                // the min frequency, the min capacity
                // will always be set to 0, except for
                // the last state. But this is okay since
                // we look at each state in order. We also
                // have to make sure that violate array
                // bounds, but this can only happen if
```

```
                    // the perf states array is badly formed
                    // or the min frequency is badly formed.
                    //
                    if (I != 0) {

                        PerfStates[I-1].MinCapacity = 0;

                    }
                    PerfStates[I].MinCapacitiy = 0;
                    Continue;

                }

                PerfState[I].MinCapacity = Total;
                Total -= Width;

        }
}
```

The logic behind this algorithm is that it is designed to allow the system to run at Lowest Voltage-Highest Frequency (henceforth the KneeState) while the battery capacity remaining is between 100% and `PopPerfDegradeThrottleMinCapacity`. Once the battery capacity falls below that level, the highest allowed state is the next-Highest Frequency available at the same voltage. When the battery capacity degrades past a certain amount (which is based on the `PopPerfDegradeThrottleMinCapacity` and the number of available states), the highest allowed state becomes the next-Highest Frequency remaining. Thus, as the battery capacity diminishes, the processor runs at a lower and lower frequency.

Another feature of this algorithm is that it allows for the OS to specify what the smallest frequency that the system can be forced to degrade to. For example, if a system supports the following frequencies at the low voltage state: 50%, 40%, 30%, 20%, and 10%, then by setting `PopPerfDegradeThrottleMinFrequency` to 30% will guarantee that we will degrade the throttle below 30%.

To minimize the amount of calculations we must make at run time, we can pre-calculate the minimum battery capacity that must be remaining to be in the indicated performance state. It should noted that any algorithm used must be able to handle where there are no more performance states after the knee state.

The beauty of this algorithm is that is can be easily replaced. If the Performance team decides that a log, exp, or any other means of calculating the "curve" is more appropriate, then only this function needs to be changed.

A sample equation could be:

$$\min Capacity = \frac{thresholdcapacity*(currentFrequency^2 - \min Frequency^2)}{(\max Frequency^2 - \min Frequency^2)}$$

4.7 System IdleLoop

To prevent the system from considering the promotion and demotion of performance states as part of how busy the system is, it is clearly desirable to invoke the increase and decrease of the CPU throttles from within the idle loop.

The basic idea of the code in the idle loop is that the code should check to see what percentage (normalized to `POP_PERF_SCALE`) of time during the last interval was spent running the idle thread and how much time was spent doing actual work. Once the percentage of actual work done is calculated, then the Operating System should look at the `PerfStates` table to see if this value falls within the operating parameters for the current bucket. It is important to make this check first because buckets are allowed to overlap each other.

If the value does not fall within the current bucket, then the operating system finds the closest performance state for which the value matches the parameters of the bucket. There is an important distinction here since by picking the nearest bucket, the operating system will have a tendency to not pick the highest or lower performance states unless there is absolutely no other choice. The advantage of this algorithm is that it will pick the `PercentFrequency` closest to the calculated value.

It is important to note that the Idle loop runs at DPC level and within the context of the processor for which it is targeted. That means that the code within the Idle loop cannot call anything that is marked as pageable. The benefit of running at DPC level is that no synchronization is required if the data structures used by the Idle loop can only be accessed by a thread running on the same processor at DPC level. This function should also be called before the C-State handler is invoked.

The algorithm would look something like this:

```
VOID
PoPerfIdle(
      PPROCESSOR_POWER_STATE  PState;
      )
{
      BOOLEAN                 C3Forced = FALSE;
      BOOLEAN                 Promoted = FALSE;
      BOOLEAN                 Demoted = FALSE;
      KPRCB                   Prcb;
      PPROCESSOR_PERF_STATE   PerfStates;
      ULONG                   TickCount;
      UCHAR                   I;
      UCHAR                   J;
      UCHAR                   CurrentPerfState
      UCHAR                   Freq;
      ULONG                   IdleTime;
      ULONG                   TickCount;
      ULONG                   Time;
      ULONG                   TimeDelta;
      ULONG                   PerfStatesCount;
```

```
//
// This piece of code should actually be done in the main
// PopIdle0 or PopProcessorIdle routines to save a function
// call. However this code is included here for completeness.
//
Prcb = CONTAINING_RECORD( PState, KPRCB, PowerState );
if (!PState->Flags & PSTATE_ADAPTIVE_THROTTLE) {
      return;
}

//
// Has enough time expired?
//
Prcb = CONTAINING_RECORD( PState, KPRCB, PowerState );
Time = POP_CUR_TIME(Prcb);
IdleTime = Prcb->IdleThread->KernelTime;
TimeDelta = Time - PState->PerfSystemTime;
if (TimeDelta < PopPerfTimeDelta) {
      return;
}

//
// Remember what the perf states are
//
PerfStates = PState->PerfStates;
PerfStatesCount = PState->PerfStatesCount;

//
// Find the bucket with the correct frequency
//
CurrentPerfState = PState->CurrentThrottleIndex;
I = CurrentPerfState;

//
// At this point, we need to see if the number of C3
// transitions have exceeded a threshold value, and if
// so, then we really need to throttle back to the
// KneeThrottleIndex since we save more power if the
// processor is at 100% and in C3 than if the processor
// is at 12.5% and in C3.
//
Freq = CalculateC3Frequency(PState);
If (Freq >= PopPerfMaxC3Frequency) {

      //
      // Set the throttle to the lowest knee in
      // the voltage & frequency curve
      //
      I = PState->KneeThrottleIndex;
      if (CurrentPerfState > I) {

            Promoted = TRUE;

      } else if (CurrentPerfState < I) {

            Demoted = TRUE;
```

```
            }

            //
            // Remember why we are doing this
            //
            C3Forced = TRUE;

            //
            // Skip to setting the throttle
            //
            goto PoPerfIdleSetThrottle;

      }

      //
      // Calculate how busy the CPU is
      //
      Freq = CalculateIdleFrequency(PState);

      //
      // Have we exceeded the thermal throttle limit?
      //
      If (Freq > PState->ThermalThrottleLimit) {

            //
            // The following code will force the frequency to
            // only as busy as the Thermal Throttle Limit will
            // actually allow. This removes the need for complicated
            // algorithms later on.
            //
            Freq = PState->ThermalThrottleLimit;
            I = PState->ThermalThrottleIndex;

      }

      //
      // Is there an upper limit to what the throttle can goto?
      // Note that because we check these after we have checked
      // the thermal limit, it means that it is not possible for
      // frequency to exceed the thermal limit that we have specified
      //
      if (PState->Flags & PSTATE_DEGRADED_THROTTLE) {

            //
            // Make sure that we don't exceed the
            // state that is specified
            //
            J = PState->ThrottleLimitIndex;
            If (Freq >= PerfStates[J].~~PercentFrequency~~IncreaseLevel) {

                  Freq = PerfStates[J].IncreaseLevel;
                  I = J;

            }

      } else if (PState->Flags & PSTATE_CONSTANT_THROTTLE) {
```

```
        J = PState->KneeThrottleIndex;
        If (Freq >= PerfStates[J].IncreaseLevelPercentFrequency) {

              Freq = PerfStates[J].IncreaseLevel;
              I = J;

        }

}

//
// Remember the adjusted value for informational purposes
// only
//
PState->LastAdjustedBusyPercentage = Freq;

//
// Find the processor frequency that best matches
// the one that we have just calculated. Please
// note that this algorithm is written in such
// a way that I can only travel in a single
// direction. It is possible to collapse the
// following code down, but not without allowing
// the possibility of I doing a "yo-yo" between
// two states (and thus never terminating the
// while-loop).
//
if (PerfState[I].IncreaseLevel < Freq) {

        If (I != 0) {

              Promoted = TRUE;
              I--;

        }

} else if (PerfStates[I].DecreaseLevel > Freq) {

        while (1)do {

              If (I==(PerfStatesCount-1)) {

                    // don't exceed the array
                    break;

              }
              Demoted = TRUE;
              I++;
              If (PerfStates[I].DecreaseLevel <= Freq) {

                    break;

              }

        } while ( PerfStates[I].DecreaseLevel > Freq);

}
```

PoPerfIdleSetThrottle:

```
        //
        // Note we need to do this now because we don't want
        // to exit this code path without having set or cancelled
        // the timer as is appropriate. The only exception to
        // this rule is in the case where the system hit the
        // C3 limit.
        //
        // Cancel the timer under the following conditions
        //
        if (I == 0) {

            //
            // We are at 100% throttle, so timer won't
            // do much of anything
            //
            KeCancelTimer(&(PState->PerfTimer));

        } else if (PState->Flags & PSTATE_CONSTANT_THROTTLE &&
            I == PState->KneeLimitIndex) {

            //
            // We are at the maximum throttle allowed
            //
            KeCancelTimer(&(PState->PerfTimer));

        } else if (PState->Flags & PSTATE_DEGRADED_THROTTLE &&
            I == PState->ThrottleLimitIndex) {

            //
            // We are at the maximum throttle allowed
            //
            KeCancelTimer(&(PState->PerfTimer));

        } else {

            //
            // No restrictions that we can think of,
            // so set the timer. Note that the semantics
            // of KeSetTimer are useful here --- if the
            // timer has already been set, then this
            // resets it (moves it to the non-signaled
            // state) and recomputes the period.
            //
            KeSetTimer(
                &PState->PerfTimer,
                ...,
                &Pstate->PerfGpe
                );

        }

        //
        // We have to make special allowances if we were forced to
        // throttle because of C3 considerations
```

```
//
if (!C3Forced) {

      //
      // See if enough time has expired to justify changing
      // the throttle. This code is here because certain
      // transitions are fairly expensive (like those across a
      // voltage state)while others are cheap. So the amount of
      // time required before we will consider promotion/demotion
      // from the expensive state might be longer than the
      // interval at which we run this function.
      //
      if ((Promoted && TimeDelta < PerfStates[I].IncreaseTime) ||
          (Demoted && TimeDelta < PerfStates[I].DecreaseTime)) {

            //
            // We haven't had enough time in the current
            // state to justify the promotion or demotion.
            // We don't update the bookkeeping since we
            // haven't considered the current interval as
            // as "success". So, we just return.
            //
            // N.B. It is very important that we don't
            // update PState->PerfSystemTime here. If we
            // did, then it is possible that TimeDelta would
            // never exceed the thresholds required.
            //
            // Base our actions for the timer upon the current
            // state instead of the target state
            //
            PopSetTimer( PState, CurrentPerfState );
            return;

      }

}

//
// At this point, we need to update the bookkeeping
//
PState->PerfIdleTime = IdleTime;
PState->PerfSystemTime = Time;
PState->PreviousC3StateTime = PState->TotalIdleState[2];

//
// Note that we need to do this now because we do not want to
// exit without having set or cancelled the timer as appropriate
//
PopSetTimer( PState, I);

//
// Update the promote and demote count
//
if (Promoted) {

      PerfStates[CurrentPerfState].IncreaseCount++;
      PState->PromotionCount++;
```

```
    } else if (Demoted) {

        PerfStates[CurrentPerfState].DecreaseCount++;
        PState->DemotionCount++;

    } else {

        //
        // At this point, we realize that we aren't
        // promoting or demoting and all the bookkeeping
        // is in order, so the appropriate thing to do
        // is just return.
        //
        PState->PerfIdleTime = IdleTime;
        PState->PerfSystemTime = Time;
        PState->PreviousC3StateTime = PState->TotalIdleState[2];
        return;

    }

    //
    // We have a new throttle. Update the bookkeeping to
    // reflect the amount of time that we spent in the
    // previous state and reset the count for the next
    // state
    //
    PopSetThrottle(
        PState,
        PerfStates,
        I,
        Time,
        IdleTime
        );
}
```

4.8 Processor Perf DPC

A desirable feature to have in an adaptive throttling mechanism is the ability to sense that the CPU has become 100% and that the throttle should be increased if required. The way to accomplish this is to schedule a periodic timer that fires if the throttle is not set to 100%.

It is important to note that this DPC may fire in situations where the CPU is not 100% busy within a given time quantum. However, since the Idle Handler resets the timer count every time it runs, the number of spurious calls to this routine should be small.

It is important to note that once the DPC has been fired, there is no need to cancel the timer since it is not scheduled as a periodic timer.

The sample algorithm would look like this:

```
VOID
PopPerfIdleAdaptiveThrottleDpc(
```

```
    IN    PKDPC Dpc,
    IN    PVOID DpcContext,
    IN    PVOID SystemArgument1,
    IN    PVOID SystemArgument2
    )
{
    PKPRCB                    Prcb;
    PKTHREAD                  IdleThread;
    PPROCESSOR_PERF_STATE     PerfStates;
    PPROCESSOR_POWER_STATE    PState;
    UCHAR                     CurrentPerfState;
    UCHAR                     Freq;
    UCHAR                     I;
    UCHAR                     J;
    ULONG                     IdleTime;
    ULONG                     Time;
    ULONG                     TimeDelta;

    //
    // We need to fetch the PRCB and the PState structures.
    // We could easily call KeGetCurrentPrcb() here, but since
    // had room for a single context, why bother making the
    // inline call (which generates more code than using the
    // context field anyways) when we can simply remember it.
    // The memory for the context field is already allocated
    // anyways.
    //
    Prcb = (PKPRCB) DpcContext;
    PState = &(Prcb->PowerState);

    //
    // Remember what the perf states are
    //
    PerfStates = PState->PerfStates;
    CurrentPerfState = PState->CurrentThrottleIndex;

    //
    // Lets see if enough kernel time has expired since
    // the last check...
    //
    Time = POP_CUR_TIME(Prcb);
    TimeDelta = Time - Pstate->PerfSystemTime;
    if (TimeDelta < PopPerfCriticalTimeTicksDelta) {

        PopSetTimer( PState, CurrentPerfState );
        return;
    }

    //
    // How much time has expired on the Idle thread?
    //
    IdleThread = Prcb->IdleThread;
    IdleTime = IdleThread->KernelTime;
    Freq = PopCalculateBusyPercentage( PState );
    TimeDelta = IdleTime - PState->PerfIdleTime;
    if (TimeDelta < PopPerfCriticalIdleTimeDelta) {
        return;
```

```
+

//
// We allow for a delta so that we can specify a range
// at which we should promote
//
Freq += (UCHAR) PopPerfCriticalFrequencyDelta;

//
// Remember which index we are currently looking at
//
I = CurrentPerfState;

//
// Have we exceeded the thermal throttle limit?
//
If (Freq > PState->ThermalThrottleLimit) {

        //
        // The following code will force the frequency to
        // only as busy as the Thermal Throttle Limit will
        // actually allow. This removes the need for complicated
        // algorithms later on.
        //
        Freq = PState->ThermalThrottleLimit;
        I = PState->ThermalThrottleIndex;

}

//
// Is there an upper limit to what the throttle can goto?
// Note that because we check these after we have checked
// the thermal limit, it means that it is not possible for
// frequency to exceed the thermal limit that we have specified
//
if (PState->Flags & PSTATE DEGRADED THROTTLE) {

        //
        // Make sure that we don't exceed the
        // state that is specified
        //
        J = PState->ThrottleLimitIndex;
        If (Freq >= PerfStates[J].PercentFrequency) {

                Freq = PerfStates[J].PercentFrequency;
                I = J;

        }

} else if (PState->Flags & PSTATE CONSTANT THROTTLE) {

        J = PState->KneeThrottleIndex;
        If (Freq >= PerfStates[J].PercentFrequency) {

                Freq = PerfStates[J].PercentFrequency;
                I = J;
        }
```

```
} else if (pState->ThermalThrottleLimit == 0) {

        //
        // This state is special --- we can only goto to
        // the fastest state if there are no thermal restrictions
        //
        I = 0;
        Freq = PerfStates[I].PercentFrequency;

}

//
// Remember this value for user information purposes
//
PState->LastAdjustedBusyPercentage = Freq;

//
// If this freq exceeds what we are currently running at,
// the we should promote, otherwise, do nothing except
// to set the timer
//
If (Freq < pState->CurrentThrottle) {

        PopSetTimer( PState, CurrentPerfState );
        Return;

}

//
// Set the timer based upon what the new state will be
//
PopSetTimer( PState, I );

//
// Update the promote count
//
if (I < CurrentPerfState) {

        PerfStates[currentPerfState].IncreaseCount++;
        PState->PromotionCount++;

} else {

        ASSERT( I < CurrentPerfState );
        Return;

}
//
// At this point, we think that the idle thread is
// stalled and that we need to do something fast to
// get it moving… Like setting it to the perf state
// that corresponds to the "knee" of the graph
// or a perf state higher than the current one…
//
if (PState->Flags & PSTATE_CONSTANT_THROTTLE) {
```

```
            //
            // Pick the knee of the curve
            //
            I = PState->KneeThrottleIndex;

      } else if (PState->Flags & PSTATE_DEGRADED_THROTTLE) {

            //
            // Pick the maximum that we are allowed to goto
            //
            I = PState->ThrottleLimitIndex;

      } else {

            //
            // Goto 100%
            //
            I = 0;

      }

      //
      // Set the new throttle
      //
      PopSetThrottle(
            PState,
            PerfStates,
            I,
            Time,
            IdleTime
            );
}
```

4.9 Setting the Watchdog Timer

To properly set the watchdog timer that must be run (in case the system becomes to busy), the following function is to be called:

```
NTSTATUS
PopSetTimer(
      IN PPROCESSOR_POWER_STATE     PState,
      IN UCHAR                      Index
      )
{
      NTSTATUS          Status;
      LARGE_INTEGER     DueTime;

      //
      // Cancel the timer under the following circumstances
      //
      if (Index == 0) {

            //
            // Already running at 100%, so we can't do anything to
            // make the computer run faster
```

```
        //
        KeCancelTimer( (PKTIMER) &(PState->PerfTimer) );
        Status = STATUS CANCELLED;

    } else if (PState->Flags & PSTATE CONSTANT THROTTLE &&
        Index == PState->KneeThrottleIndex) {

        //
        // We are already running at the maximum constant
        // throttle allowed
        //
        KeCancelTimer( (PKTIMER) &(PState->PerfTimer) );
        Status = STATUS CANCELLED;

    } else if (PState->Flags & PSTATE DEGRADED THROTTLE &&
        Index == PState->ThrottleLimitIndex) {

        //
        // We are already running at the maximum degraded
        // throttle allowed
        //
        KeCancelTimer( (PKTIMER) &(PState->PerfTimer) );
        Status = STATUS CANCELLED;

    } else {

        //
        // No restrictions that we can think of, so set the
        // timer. Note that the semantics are useful here --- if
        // the timer has already been set, then this resets
        // it (moves it back to the non-signaled state) and
        // recomputes the period
        //
        DueTime.QuadPart = -1 * PopPerfCriticalTimeDelta;
        KeSetTimer(
              (PKTIMER) &(PState->PerfTimer),
              dueTime,
              &(PState->PerfDpc)
              );
        status = STATUS SUCCESS

    }
    return status;
}
```

5.0 Initialization Changes

5.1 Global Variable Initialization

The following global variables must be initialized at the same time that the kernel is loaded. To simplify changing these values later on, the Kernel will provide some default values that can be overridden by the registry.

- PopPerfTimeDelta: A value in ~~the same units as PRCB->KernelTime~~microseconds that corresponds to the Time Delta that must have

occurred before the Idle thread will attempt to determine how busy the CPU was during the previous interval. The kernel will convert this number to the same units as PRCB->KernelTime in a variable known as PopPerfTimeTicks.

- PopPerfCriticalTimeDelta: A value in ~~the same units as PRCB->KernelTime~~microseconds that corresponds to the Time Delta that must have occurred before the Timer DPC will attempt to determine how busy the CPU was during the previous interval. The kernel will convert this number to the same units as PRCB->KernelTime in a variable known as PopPerfCriticalTimeTicks.

~~PopPerfCriticalIdleTimeDelta: A value in the same units of PRCB->KernelTime that corresponds to the Time Delta that must have occurred for the IdleThread during a PopPerfCriticalTimeDelta period. If this time has not occurred, then the throttle will be raised by the TimerDPC.~~

- PopPerfCriticalFrequencyDelta. This is a percentage to add when calculating CPU busyness in the watchdog timer. This value will allow for a faster triggering of the watchdog, under ligher loads. The recommended value is 0.
- PopPerfIncreasePercentModifier: A value between 0 and 100 where lower means that overall IncreaseLevel value will be higher (and thus promotions won't occur as frequently) that indicates what percentage of the delta between the current state and the state to promote to should be used to set the promote level. A suggested value would be 0%
- PopPerfIncreaseAbsoluteModifier: A value between 0 and 100 where lower means that the overall IncreaseLevel value will be higher (and thus promotions won't occur as frequently) that indicates how many extra percentage points to remove to the promote level. It should be noted that if the value is particularly high, then it might not be possible to promote from this state. A suggested value would be 1%.
- PopPerfDecreasePercentModifier: A value between 0 and 100 where higher means that overall DecreaseLevel value will be lower (and thus demotions won't occur as frequently) that indicates what percentage of the delta between the current state and the state to demote to should be used to set the demote level. A suggested value would be 50%
- PopPerfDecreaseAbsoluteModifier: A value between 0 and 100 where higher means that the overall DecreaseLevel value will be lower (and thus demotions won't occur as frequently) that indicates how many extra percentage points to subtract to the demote level. It should be noted that if the value is particularly high, then it might not be possible to demote from this state. A suggested value would be 1%.
- PopPerfIncreaseTimeValue: A value in ~~the same units as PRCB->KernelTime~~microseconds that corresponds to the Time Delta that must have occurred before a Throttle Increase is considered. This value should be in multiple of PopPerfTimeDelta. This value may also serve a basis for calculating different time increments for each Throttle Step.
- PopPerfIncreaseMininimumTime. A value in microseconds that corresponds to the absolute minimum amount of time since a promotion has occurred before another one is allowed. A recommended value is 300 ms.

- PopPerfDecreaseTimeValue: A value in ~~the same units as PRCB->KernelTime~~microseconds that corresponds to the Time Delta that must have occurred before a Throttle Decrease is considered. This value should be in multiple of PopPerfTimeDelta. This value may also serve a basis for calculating different time increments for each Throttle Step.
- PopPerfDecreaseMinimumTime. A value in microseconds that corresponds to the absolute minimum amount of time since a demotion has occurred before another one is allowed. A recommended value is 1000 ms.

~~PopProcPerfStateLookAsideList: This is a lookaside list that will be used to allocate PROCESSOR_PERF_STATE structures.~~

- PopPerfDegradeThrottleMinCapacity: A value between 0 and 100 that represents at what point of battery capacity we will start forcing down the throttle when we are in the Degraded Throttling mode. For example, a value of 50% means that we will start throttling when the CPU reaches 50%.
- PopPerfDegradeThrottleMinFrequency: A value between 0 and 100 that represents the lowest frequency that we can force the throttle down to when we are start the Degraded Throttling mode. For example, a value of 30% means that we will force the throttle below 30%.
- PopPerfMaxC3Frequency: A percentage value that represents the maximum amount of time that was spend in C3 for the last quanta before the idle look will decide that it should optimize for C3 usage. A sample value would be 50%.

5.2 PoInitializePrcb()

The PROCESSOR_POWER_STATE structure is initialized by PoInitializePrcb(). The following changes must occur:

```
VOID
FASTCALL
PoInitializePrcb(
      PKPRCB      Prcb
      )
{
      //
      // Zero power state structure
      //
      RtlZeroMemory(&Prcb->PowerState,sizeof(Prcb->PowerState));

      //
      // Initialize to legacy functions with promotion from it disabled
      //
      Prcb->PowerState.Idle0KernelTimeLimit = (ULONG) -1;
      Prcb->PowerState.IdleFunction = PopIdle0;
      Prcb->PowerState.CurrentThrottle = POP_PERF_SCALE;

      //
      // Calculate this value exactly once
      //
      KeQueryPerformanceCounter(
            &Prcb->PowerState.PerfCounterFrequency
            );
```

```
        //
        // Initialize the Adaptive throttling subcomponents
        //
        KeInitializeDpc(
              &(Prcb->PowerState.PerfDpc),
              PopPerfAdaptiveThrottleDpc,
              PrcbNULL
              );
        KeSetTargetProcessorDpc(
              &(Prcb->PowerState.PerfDpc),
              Prcb->Number
              );
        KeInitalizeTimer(
              &(Prcb->PowerState.PerfTimer)
              );
}
```

5.3 PopSetPerfLevels()

The PROCESSOR_PERF_STATE structure is initialized by calls to PopSetPerfLevels(). The initialization can occur as before, when it was initializing PROCESSOR_PERF_LEVEL instead.

```
NTSTATUS
PopSetPerfLevels(
      IN    PPROCESSOR_STATE_HANDLER2 ProcessHandler
      )
{
      BOOLEAN                    FailedAllocation = FALSE;
      KAFFINITY                  Processors, CurrentAffinity;
      KIRQL                      OldIrql;
      PKPRCB                     Prcb;
      NTSTATUS                   Status = STATUS_SUCCESS;
      ULONG                      i;
      ULONG                      PerfStatesCount = 0;
      UCHAR                      Freq;
      UCHAR                      KneeThrottleIndex = 0;
      UCHAR                      MinThrottle;
      UCHAR                      MaxThrottle;
      UCHAR                      ThermalThrottleIndex = 0;
      PPROCESSOR_PERF_STATE      PerfStates = NULL;
      PPROCESSOR_PERF_STATE      TempStates;
      PPROCESSOR_POWER_STATE     PState;

      //
      // The first step is to convert the data that was passed to us
      // PROCESSOR_PERF_LEVEL over to PROCESSOR_PERF_STATE.
      //
      if (ProcessorHandler->NumPerfStates) {

            //
            // Because we are using going to allocate the PerfStates
            // array first so that that we can work on it, then copy
            // it to each processor, we mustean do that allocation from
            // non-paged pool.
```

```
//
PerfStatesCount = ProcessorHandler->NumPerfStates;
PerfStates = ExAllocatePoolWithTag(
      NonPagedPool,
      PerfStatesCount * sizeof(PROCESSOR_PERF_STATE),
      'sPoP'
      );
if (PerfStates == NULL) {

      //
      // We can handle this case. We will set the
      // status code to an appropriate failure code
      // and we will clean up the existing processor
      // states. The reason we do that is because
      // this function only gets called if the current
      // states are invalid, so keeping the current ones
      // would make no sense.
      //
      sStatus = STATUS_INSUFFICIENT_RESOURCES;
      PerfStateCount = 0;
      goto PopSetPerfLevelsSetNewStates;

}
RtlZeroMemory(
      PerfStates,
      PerfStatesCount * sizeof(PROCESSOR_PERF_STATE)
      );

//
// Initialize each of the PROCESSOR_PERF_STATE entries
//
for (i = 0; i < PerfStatesCount; i++) {

      PerfStates[i].PercentFrequency =
            ProcessorHandler->PerfLeve[i].PercentFrequency;
      PerfStates[i].Power =
            ProcessorHandler->PerfLevel[i].Power;

}

//
// Analyze the PerfStates to determine which entries are
// linear/non-linear
//
PopAnalyzePerfStates( PerfStates, PerfStatesCount );

//
// Calculate the increase level, decrease level, and
// increase/decrease time
//
PopCalculateIncreaseLevel( PerfStates, PerfStatesCount );
PopCalculateDecreaseLevel( PerfStates, PerfStatesCount );
PopCalculateMinCapacity( PerfStates, PerfStatesCount );
PopCalculateIncreaseDecreaseTime(
      PerfStates,
      PerfStatesCount,
      PerfHandler
```

```
        );

    //
    // Calculate where the Knee in the performance curve is
    //
    for (i=PerfStatesCount; i >= 1; i++) {

        if (PerfStates[i-1].Flags & POP_THROTTLE_NON_LINEAR){

            KneeThrottleIndex = i-1;
            Break;

        }

    }

    //
    // Find the minimum throttle value which is greather than
    // the specified default and the current max throttle
    //
    MinThrottle = POP PERF SCALE;
    MaxThrottle = 0;
    For (I = 0; I < PerfStatesCount; I++) {

        Freq = PerfStates[I].PercentFrequency;
        If (Freq < MinThrottle &&
            Freq >= PopIdleDefaultMinThrottle) {

            MinThrottle = Freq;

        }
        If (Freq > MaxThrottle &&
            Freq >= PopIdleDefaultMinThrottle) {

            MaxThrottle = Freq;
            ThermalThrottleIndex = (UCHAR) I;

        }

    }

    //
    // Make sure that we can run
    //
    ASSERT( MaxThrottle >= PopIdleDefaultMinThrottle );

}

PopSetPerfLevelsSetNewStates:

    If (!PerfStates) {

        //
        // Remember that our min and max are 100%
        //
        MinThrottle = MaxThrottle = POP PERF SCALE;
```

```
}

if (PerfStates) {

        //
        // We have perf states, so remember that in our
        // capabilities
        //
        PopCapabilities.ProcessorThrottle = TRUE;

        //
        // Find the minimum throttle value >=
        // PopIdleDefaultMinThrottle and the current maximum
        // throttle value
        //
        PopCapabilities.ProcessorMinThrottle = POP_PERF_SCALE;
        PopCapabilities.ProcessorMaxThrottle = 0;
        for (i=0;i<ProcessorHandler->NumPerfStates;i++) {
                Freq = PerfStates->PerfLevel[i].PercentFrequency;
                if ((Freq < PopCapabilities.ProcessorMinThrottle) &&
                        (Freq >= PopIdleDefaultMinThrottle)) {

                        PopCapabilities.ProcessorMinThrottle = Freq;

                }
                if ((Freq > PopCapabilities.ProcessorMaxThrottle) &&
                        (Freq >= PopIdleDefaultMinThrottle)) {

                        PopCapabilities.ProcessorMaxThrottle = Freq;
                        ThermalThrottleIndex = i;

                }
        }

        //
        // There better be SOME speed we can run at.
        //
        ASSERT(PopCapabilities.ProcessorMaxThrottle >=
                PopIdleDefaultMinThrottle);

} else {

        //
        // We don't have any perf sates, so remember that in our
        // capabilities
        //
        PopCapabilities.ProcessorThrottle = FALSE;
        PopCapabilities.ProcessorMaxThrottle = POP_PERF_SCALE;
        PopCapabilities.ProcessorMinThrottle = POP_PERF_SCALE;
}

//
// Initialize the PPROCESSOR_POWER_STATE for each processor
//
Processors = KeActiveProcessors;
CurrentAffinity = 1;
```

```
while (Processors) {

      if (!(Processors & CurrentAffinity) {

            CurrentAffinity <<= 1;
            Continue;

      }

      //
      // Remember that we did this processor and make sure that
      // we are actually running on that processor. This ensures
      // that we are synchronized with the DPC and idle loop
      // routines
      //
      Processors &= ~CurrentAffinity;
      KeSetSystemAffinityThread(CurrentAffinity);
      CurrentAffinity <<= 1;

      //
      // To make sure that we aren't pre-empted, we must raise
      // to DISPATCH_LEVEL
      //
      KeRaiseIrql(DISPATCH_LEVEL, &OldIrql );

      //
      // Get the PRCB and PPROCESSOR_POWER_STATE structures that
      // we will need to manipulate
      //
      Prcb = KeGetCurrentPrcb();
      PState = &Prcb->PowerState;

      //
      // Remember what our thermal limit is
      //
      PState->ThermalThrottleLimit = MaxThrottle;
            PopCapabilities.ProcessorMaxThrottle;
      PState->ThermalThrottleIndex = ThermalThrottleIndex;

      //
      // Likewise, remember what the min and max throttle are
      //
      PState->ProcessorMinThrottle = MinThrottle;
      PState->ProcessorMaxThrottle = MaxThrottle;

      //
      // To get the bookkeeping to work out correctly, we will
      // set the throttle to 0% (which is not possible), set the
      // current index to the last state, and set current tick
      // count to the current time.
      //
      PState->CurrentThrottle = 0;
      PState->PerfTickCount = POP_CUR_TIME(Prcb)Time;
      If (PerfStatesCount) {

            PState->CurrentThrottleIndex = PerfStatesCount - 1;
```

```
    } else {

        PState->CurrentThrottleIndex = 0;

    }

    //
    // Reset the Knee Index. This indicates where the knee
    // in the performance curve is
    //
    PState->KneeThrottleIndex = KneeThrottleIndex;

    //
    // Reset the Throttle Limit Index
    //
    PState->ThrottleLimitIndex = KneeThrottleIndex;

    //
    // Reset these value since it doesn't make much sense to
    // keep track of these across state changes
    //
    PState->PromotionCount = 0;
    PState->DemotionCount = 0;

    //
    // Reset the values to something that makes sense. We can
    // assume that we started at 100% busy and 0% C3 Idle
    //
    PState->LastBusyPercentage = 100;
    PState->LastAdjustedBusyPercentage = 100;
    PState->LastC3Percentage = 0;

    //
    // If there are already perf states present for this
    // processor, then free them
    //
    if (PState->PerfStates) {

        ExFreeToNPagedLookasideList(
            &PopProcPerfStateLookAsideList,
            PState->PerfStates
            );
        ExFreePool( PState->PerfStates );
        PState->PerfStates = NULL;
        PState->PerfStatesCount = 0;

    }

    //
    // At this point, we have to distinguish our behavior based
    // on whether or not we have perf states
    //
    if (PerfStates) {

        //
        // We do, so let allocate some memory and make a copy
        // of the template that we already created.
```

```
//
TempStates = ExAllocateFromNPagedLookasideList(
      &PopProcPerfStateLookAsideList
      );
TempStates = ExAllocatePoolWithTag(
      NonPagedPool,
      PerfStatesCount * sizeof(PROCESSOR_PERF_STATE),
      'sPoP'
      );
if (TempStates == NULL) {

      //
      // Not being able to allocate this structure
      // is surely fatal. The only way to get around
      // it (I think) is to break out of this case
      // and treat it as if there are no PerfStates
      // available.
      //
      status = STATUS_INSUFFICIENT_RESOURCES;
      failedAllocation = TRUE;

      PState->Flags &= ~PSTATE_SUPPORTS_THROTTLE;
      PState->PerfSetThrottle = NULL;
      KeLowerIrql( oldIrql );
      Continue;
      KeBugCheckEx(
            INTERNAL_POWER_FAILURE,
            6,
            STATUS_INSUFFICIENT_RESOURCES,
            __LINE__,
            0
            );

} else {

      //
      // Copy the template to the one associated with
      // the processor
      //
      RtlCopyMemory(
            TempStates,
            PerfStates,
            PerfStatesCount *
                  sizeof(PROCESSOR_PERF_STATES)
            );
      PState->PerfStates = TempStates;
      PState->PerfStatesCount = PerfStatesCount;

}

//
// Remember that we support throttling
//
PState->Flags = PSTATE_SUPPORTS_THROTTLE;
PState->PerfSetThrottle =
      ProcessorHandler->SetPerfLevel;
```

```
            //
            // Update the processor status now
            //
            PopUpdateProcessorThrottle();

        } else {

            //
            // Remember that we don't support throttling
            //
            PState->Flags = 0;
            PState->PerfSetThrottle = NULL;

        }

        //
        // Update the processor throttle function
        //
        if (PState->PerfStates) {

            PState->PopSetThrottle =
                    ProcessorHandler->SetPerfLevel;

        } else {

            PState->PopSetThrottle = NULL;

        }

        //
        // Update the processor throttle (since we are already
        // running on the target processor
        //
        PopUpdateProcessorThrottle();

        //
        // We can now return to our previous IRQL
        //
        KeLowerIrql( OldIrql );
    }

    //
    // Did we fail an allocation and thus require a cleanup?
    //
    if (FailedAllocation) {

        processors = KeActiveProcessors;
        currentAffinity = 1;
        while (processors) {

            if (!processors & currentAffinity)) {

                currentAffinity <<= 1;
                continue;

            }
```

```
processors &= ~currentAffinity;
KeSetSystemAffinityThread( currentAffinity );
CurrentAffinity <<= 1;

KeRaiseIrql( DISPATCH LEVEL, &OldIrql );

Prcb = KeGetCurrentPrcb();
PState = &(Prcb->PowerState);

//
// Reset the PowerState
//
PState->ThermalThrottleLimit = POP PERF SCALE;
PState->ThermalThrottleIndex = 0;
PState->ProcessorMinThrottle = POP PERF SCALE;
PState->ProcessorMaxThrottle = POP PERF SCALE;
PState->CurrentThrottle      = POP PERF SCALE;
PState->PerfTickCount        = POP CUR TIME(Prcb);
PState->CurrentThrottleIndex = 0;
PState->KneeThrottleIndex    = 0;
PState->ThrottleLimitIndex   = 0;

//
// Free Allocated structures if any
//
if (PState->PerfStates) {

      maxThrottle =
            PState->PerfStates[0].PercentFrequency;
      ExFreePool( PState->PerfStates );

} else {

      maxThrottle = POP PERF SCALE;

}
pState->PerfStates = NULL;
pState->PerfStatesCount = 0;

//
// Return to 100% if possible
//
if (PState->PerfSetThrottle) {

      PState->PerfSetThrottle( maxThrottle );

}

PState->Flags = 0;
PState->PerfSetThrottle = NULL;

//
// Return to previous IRQL
//
KeLowerIrql( OldIrql );
}
```

```
        //
        // Set the global caps
        //
        PopCapabilities.ProcessorThrottle = FALSE;
        PopCapabilities.ProcessorMinThrottle = POP_PERF_SCALE;
        PopCapabilities.ProcessorMaxThrottle = POP_PERF_SCALE;

    } else {

        //
        // Remember these caps
        //
        PopCapabilities.ProcessorThrottle (PerfStates != NULL);
        PopCapabilities.ProcessorMinThrottle = minThrottle;
        PopCapabilities.ProcessorMaxThrottle = maxThrottle;

    }

    //
    // Return to the proper affinity
    //
    KeRevertToUserAffinityThread();

    //
    // Free the memory we used
    //
    If (PerfStates) {

        ExFreePool( PerfStates );

    }
        //
        // Return whatever status code we have
        //
        return status;
}
```

This function has been extensively changed from the existing code base. The first and most obvious change is the fact that we allocate per-processor perf state arrays. This means that we have to be able to handle the case where we cannot allocate such an array. The solution to this problem is to basically treat that failure as an inability of the system to throttle.

The changes to this function also means that a great deal of additional intelligence will have to be present in `PopUpdateProcessorThrottle()`.

The purpose of this algorithm is to flush each processor from using the old processor state tables. This is accomplished by running the thread in the context of each processors. Since this structure is only accessed within the context of a `TimerDPC` and the `IdleThread`, which both run at DPC level, and thus cannot be pre-empted, this guarantees that neither the `IdleThread` nor the `TimerDPC` can be running. If neither

are running, then neither are using the old copy of the structure, which means that it can be safely freed and the new one used in its place.

The only potential problem with this algorithm is dealing with a potential low memory situation. It is not acceptable to share copies of the `PerfStates` structure among multiple processors. That leaves as solutions either guaranteeing that we don't run into low memory problem or issuing a bugcheck if we do. To avoid running into low memory problems, the `PerfStates` structures can be allocated from an `NPAGED_LOOKASIDE` list. A further optimization is that if the old `PerfStates` structure is the same size or larger than the new one, it could be used instead.

5.4 PopUpdateAllThrottles()

This function is used to update all the throttles simultaneously.

```
VOID
PopUpdateAllThrottles(
      VOID
      )
{
      KAFFINITY   Processors;
      KAFFINITY   CurrentAffinity;
      KIRQL       OldIrql;
      KPRCB       PState;

      Processors = KeActiveProcessors;
      CurrentAffinity = 1;
      while (Processors) {

            if (Processors & CurrentAffinity) {

                  Processors &= ~CurrentAffinity;
                  KeSetSystemAffinityThread(CurrentAffinity);

                  //
                  // We must call PopUpdateProcessorThrottle
                  // at DISPATCH_LEVEL
                  //
                  KeRaiseIrql( DISPATCH_LEVEL, &OldIrql );
                  PState = &(KeGetCurrentPrcb()->PowerState);
                  If (PState->Flags & PSTATE SUPPORTS THROTTLES) {

                        PopUpdateProcessorThrottle();

                  }
                  KeLowerIrql( OldIrql );

            }
            CurrentAffinity <<= 1;

      }
      KeRevertToUserAffinityThread();
}
```

The principle change to this code is that we no longer have an early exit case if there are no perf states registered. In theory, we could add a check that would look at `PopCapabilities.ProcessorThrottle`.

The other change is that we will always call `PopUpdateProcessorThrottle` at `DISPATCH_LEVEL`. This will guarantee that the DPC routine will not be able to pre-empt the routine.

5.5 PopUpdateProcessorThrottles()

```
VOID
PopUpdateProcessorThrottles(
      VOID
      )
{
      PKRPRCB                   Prcb;
      PPROCESSOR_PERF_STATE     PerfStates;
      PPROCESSOR_POWER_STATE    PState;
      UCHAR                     I;
      UCHAR                     Index;
      UCHAR                     NewLimit;
      UCHAR                     PerfStatesCount;
      ULONG                     IdleTime
      ULONG                     Time;

      //
      // Get the PowerState structure from the PRCB
      //
      Prcb = KeGetCurrentPrcb();
      PState = &Prcb->PowerState;

      //
      // Make sure that this processor supports throttling
      //
      If (PState->PopSetThrottle == NULL) {

            Return;

      }

      //
      // Get the current information such as curren throttle, current
      // throttle index, current system time, current idle time
      //
      NewLimit = PState->CurrentThrottle;
      Index = PState->CurrentThrottleIndex;
      Time = CUR_TIME(Prcb);
      IdleTime = Prcb->IdleThread->KernelTime;

      //
      // We will need to refer to these frequently
      //
      PerfStates = PState->PerfStates;
```

```
PerfStatesCount = PState->PerfStatesCount;

//
// If we are on AC, then we always want to run at the highest
// possible speed. Also the same algorithm is used on DC if the
// dynamic throttling policy is PO_THROTTLE_NONE.
//
if ((PopPolicy == &PopAcPolicy) ||
      (PopPolicy->DynamicThrottle == PO_THROTTLE_NONE)) {

      //
      // We precompute what the max throttle should be
      //
      Index = PState->ThermalThrottleIndex;
      NewLimit = PerfStates[Index].PercentFrequency;

} else {

      //
      // We are on DC, apply the appropriate heuristics based on
      // the dynamic throttling policy.
      //
      switch (PopPolicy->DynamicThrottle) {
      case PO_THROTTLE_CONSTANT:

            //
            // We have pre-computed the optimal point on the
            // already. So, we might as well use that.
            //
            Index = PState->KneeThrottleIndex;
            NewLimit = PerfStates[Index].PercentFrequency;

            //
            // Set the constant flag and clear the degraded flag
            //
            PState->Flags &= ~PSTATE_DEGRADED_THROTTLE;
            PState->Flags |= PSTATE_CONSTANT_THROTTLE;
            PState->Flags |= PSTATE_ADAPTIVE_THROTTLE;
            break;

      case PO_THROTTLE_DEGRADE:

            //
            // We calculate the limit of the degrade throttle
            // on the fly.
            //
            Index = PState->ThrottleLimitIndex;
            NewLimit = PerfStates[Index].PercentFrequency;

            //
            // Set the degraded flag and clear the constant flag
            //
            PState->Flags &= ~PSTATE_CONSTANT_THROTTLE;
            PState->Flags |= PSTATE_DEGRADEDCONSTANT_THROTTLE;
            PState->Flags |= PSTATE_ADAPTIVE_THROTTLE;
            break;
```

```
        case PO_THROTTLE_ADAPTIVE:

              PState->Flags |= PSTATE_ADAPTIVE_THROTTLE;
              break;

        default:
              // not implemented
              PoPrint(
                    PO_THROTTLE,
                    ("PopUpdateProcessorThrottle - unimplemented"
                    " dynamic throttle %d\n",
                    PopPolicy->DynamicThrottle)
                    );
              break;
        }
}

//
// Check if we are over the thermal limit.
//
ASSERT(PState->ThermalThrottleLimit >=
      PopCapabilities.ProcessorMinThrottle);
if (NewLimit > PState->ThermalThrottleLimit) {

      PoPrint(
            PO_THERM,
            ("PopUpdateProcessorThrottle - new throttle limit %d"
            " over thermal limit %d\n",
            NewLimit,
            PState->ThermalThrottleLimit)
            );
      NewLimit = PState->ThermalThrottleLimit;
      Index = PState->ThermalThrottleIndex;

}

//
// Apply new throttle if it has changed.
//
if (NewLimit != PState->CurrentThrottle) {

      PoPrint(
            PO_THROTTLE,
            ("PopUpdateProcessorThrottle - Setting CPU throttle "
            "to %d\n",NewLimit)
            );
      if (newLimit < PState->CurrenThrottle) {

            PState->DemotionCount++;
            PerfStates[PState->CurrentThrottleIndex].
                  DecreaseCount++;

      } else {

            PState->PromotionCount++;
            PerfStates[PState->CurrenThrottleIndex].
                  IncreaseCount++;
```

```
                }
                _____________
                PopSetThrottle(
                        PState,
                        PerfState,
                        Index,
                        Time,
                        IdleTime
                        );

        }

}
```

It should be noted that this function can only be called within the context of the target processor. This function does not acquire any spinlocks because it is running at DISPATCH_LEVEL, thus preventing the Timer DPC and the Idle Thread from running on this processor.

5.5 PopApplyThermalThrottle()

```
VOID
PopApplyThermalThrottle(
        VOID
        )
{
        //
        // <Code which is not relevant to this spec here>
        //
#if DBG
        PoPrint(
                PO_THERM,
                ("Thermal - Zone %p - %s - Thermal throttle = %d.%dn",
                thermalZone, t,
                (thermalThrottle / 10),
                (thermalThrottle % 10))
                );
        PoPrint(
                PO_THERM,
                ("Thermal - Zone %p - %s - Forced throttle = %d.%d\n",
                thermalZone, t,
                (forcedThrottle / 10),
                (forcedThrottle % 10))
                );
#endif

        //
        // Set limit on effected processors
        //
        processorNumber = 0;
        currentAffinity = 1;
        processors = KeActiveProcessors;

        do {
```

```
if (~(processors & currentAffinity) {

      currentAffinity <<= 1;
      continue;

}

processors &= ~currentAffinity;

//
// We must run on the target processor
//
KeSetSystemAffinityThread(currentAffinity);

pState = &(KeGetCurrentPrcb()->PowerState);

//
// We need to be running at DISPATCH_LEVEL to access the
// structures referenced within the pState…
//
KeRaiseIrql( DISPATCH_LEVEL, &OldIrql );
pState = &(KeGetCurrentPrcb()->PowerState);

if ((pState->Flags & PSTATE SUPPORTS THROTTLE) == 0) {

      CurrentAffinity <<= 1;
      KeLowerIrql( OldIrql );
      Continue;

}
```

```
            //
            // Convert throttles to processor bucket size. We need to
            // do this in the context of the target processor processor
            // to make sure sure that we get correct set of perf levels
            //
            PopRoundThrottle(
                  (UCHAR)(thermalThrottle/PO_TZ_THROTTLE_SCALE),
                  &thermalLimit,
                  NULL,
                  &thermalLimitIndex,
                  NULL
                  );
            PopRoundThrottle(
                  (UCHAR)(forcedThrottle/PO_TZ_THROTTLE_SCALE),
                  &forcedLimit,
                  NULL,
                  &forcedLimitIndex,
                  NULL
                  );
#if DBG
            PoPrint(
                  PO_THERM,
                  ("Thermal - Zone %p - %d - Thermal Limit = %d\n",
                  thermalZone, Prcb->Number, thermalLimit)
                  );
```

```
                PoPrint(
                      PO_THERM,
                      ("Thermal - Zone %p - %d - Forced Limit = %d\n",
                      thermalZone, Prcb->Number, forcedLimit)
                      );
#endif

                //
                // Figure out which one we are going to use
                //
                limit = (thermalProcessors & currentAffinity) ?
                      thermalLimit : forcedLimit;
                index = (thermalProcessors & currentAffinity) ?
                      thermalLimitIndex : forcedLimitIndex;

                //
                // Next affinity mask
                //
                currentAffinity <<= 1;

                //
                // Does this processor support throttling?
                //
                if (pState->PopSetThrottle == NULL) {

                      KeLowerIrql( OldIrql );
                      continue;

                }

                //
                // Check processors limit for a change
                //
                if (limit > PState->opCapabilities.ProcessorMaxThrottle) {

                      PoPrint(
                            PO_THERM,
                            ("PopThrottle: Limit (%d) > Scale (%d)\n",
                            limit,
                            PopCapabilities.ProcessorMaxThrottle)
                            );
                      limit = PState->opCapabilities.ProcessorMaxThrottle;

                } else if (limit < PState-
                >opCapabilities.ProcessorMinThrottle) {

                      PoPrint(
                            PO_THERM,
                            ("PopThrottle: Limit (%d) < MinThrottle "
                            "(%d)\n",
                            limit,
                            PopCapabilities.ProcessorMinThrottle)
                            );
                      limit = PState->opCapabilities.ProcessorMinThrottle;

                }
```

```
            if (pState->ThermalThrottleLimit != limit) {

                pState->ThermalThrottleLimit = limit;
                pState->ThermalThrottleIndex = index;

            }

            //
            // Revert back to our previous IRQL
            //
            KeLowerIrql( OldIrql );

        } while (processors);

        KeRevertToUserAffinityThread();

        //
        // Apply thermal throttles if necessary. Note we always do this
        // whether or not the limits were changed. This routine also gets
        // called whenever the system transitions from AC to DC, and that
        // may also require a throttle update due to dynamic throttling.
        //
        PopUpdateAllThrottles();
}
```

5.6 PopRoundThrottle()

```
VOID
PopRoundThrottle(
        IN UCHAR Throttle,
        OUT OPTIONAL PUCHAR RoundDown,
        OUT OPTIONAL PUCHAR RoundUp,
        OUT OPTIONAL PUCHAR RoundDownIndex,
        OUT OPTIONAL PUCHAR RoundUpIndex
        )
{
        KIRQL                   OldIrql;
        PKPRCB                  Prcb;
        PPROCESSOR_PERF_STATE   PerfStates;
        PPROCESSOR_POWER_STATE  Pstate;
        UCHAR                   Low;
        UCHAR                   LowIndex;
        UCHAR                   High;
        UCHAR                   HighIndex;
        ULONG                   I;

        //
        // We need to get this processor's power capabilities
        //
        Prcb = KeGetCurrentPrcb();
        PState = &(Prcb->PowerState);

        //
        // Make sure that we are synchronized with the Idle thread
        // and other routines that access these data structures
```

```
//
KeRaiseIrql( DISPATCH_LEVEL, &OldIrql );
PerfStates = PState->PerfStates;

//
// Does this processor support throttling?
//
if (!(PState->Flags & PSTATE_SUPPORTS_THROTTLE)PopSetThrottle ==
NULL) {

      if (ARGUMENT_PRESENT(RoundUp)) {

            *RoundUp = Throttle;
            if (ARGUMENT_PRESENT(RoundUpIndex)) {

                  *RoundUpIndex = 0;

            }

      }
      if (ARGUMENT_PRESENT(RoundDown)) {

            *RoundDown = Throttle;
            if (ARGUMENT_PRESENT(RoundDownIndex)) {

                  *RoundDownIndex = 0;

            }

      }
      KeLowerIrql( OldIrql );
      return;

      Low = High = Throttle;
      LowIndex = HighIndex = 0;
      Goto PopRoundThrottleExit;

}

//
// Check if the supplied throttle is out of range
//
if (Throttle <= PopCapabilities.State->ProcessorMinThrottle) {

      Throttle = PState->opCapabilities.ProcessorMinThrottle;

} else if (Throttle >= PopCapabilities.State-
>ProcessorMaxThrottle) {

      Throttle = PState->opCapabilities.ProcessorMaxThrottle;

}

//
// Initialize our search space to something reasonable
//
Low = High = PerfStates[0].PercentFrequency;
```

```
LowIndex = HighIndex = 0;

//
// Look at all the available perf states
//
for ( I = 0; I < PState->PerfStatesCountPopPerfLevelCount; I++) {

      if (Low > Throttle) && {
            (PerfStates[I].PercentFrequency < Low)) {

            if (PerfStates[I].PercentFrequency < low) {

                  Low = PerfStates[I].PercentFrequency;
                  LowIndex = I;

            }

      } else if (ThrottlePerfStates[I].PercentFrequency > Low) {

            if (PerfStates[I].PercentFrequency <= Throttle &&
                PerfStates[I].PercentFrequency > low) {

                  Low = PerfStates[I].PercentFrequency;
                  LowIndex = I;

            }

      }

            if (High < Throttle) {&&
                  (PerfStates[I].PercentFrequency -> High)) {

                  if (PerfStates[I].PercentFrequency > high) {

                        High = PerfStates[I];
                        HighIndex = I;

                  }

            } else if (ThrottlePerfStates[I].PercentFrequency < High) {

                  if (PerfStates[I].PercentFrequency >= Throttle &&
                      PerfStates[I].PercentFrequency < High) {

                        High = &PerfStates[I];
                        HighIndex = I;

                  }

            }

      }

PopRountThrottleExit:

      //
      // Revert back to our previous IRQL.
```

```
        //
        KeLowerIrql( OldIrql );

        //
        // Fill in the pointers provided by the caller
        //
        if (ARGUMENT_PRESENT(RoundUp)) {

              *RoundUp = High;
              if (ARGUMENT_PRESENT(RoundUpIndex)) {

                    *RoundUpIndex = HighIndex;

              }

        }

        if (ARGUMENT_PRESENT(RoundDown)) {

              *RoundDown = Low;
              if (ARGUMENT_PRESENT(RoundDownIndex)) {

                    *RoundDownIndex = LowIndex;

              }
        }
  }
```

The changes in this routine include an optimization for dealing with the case where the desired throttle is above/below the maximum/minimum. It also now returns the index into the `PerfStates` array that correspond with the rounded up and rounded down values.

5.7 PopCompositeBatteryDeviceHandler()

This routine is the one that is notified when the total battery remaining level changes. For adaptive throttling to work, whenever a new battery notification comes in, we need to update the current ThrottleLimitIndex.

```
VOID
PopCompositeBatteryDeviceHandler(
      IN PDEVICE_OBJECT DeviceObject,
      IN PIRP           Irp,
      IN PVOID          Context
      )
{
      <...>

      if (NT_SUCCESS(Irp->IoStatus.Status) {

            //
            // Handle the completed request
            //
            switch (PopCB.State) {
            <...>
            case PO_CB_READ_STATUS:
```

```
<...>
if (Policy == &PopDCPolicy) {

      <...>

      //
      // This is kind of silly, but since we
      // want to minimize our synchronization
      // elsewhere, we have to examine every
      // processor's PowerState and update the
      // ThrottleLimitIndex on each. This
      // may eventually be the smart thing to
      // do if not all processors support the
      // same set of states.
      //
      currentAffinity = 1;
      processors = KeActiveProcessors;
      while (processors) {

            if (!(processors & currentAffinity)) {

                  currentAffinity <<= 1;
                  continue;

            }

            KeSetSystemThreadAffinity(
                  currentAffinity
            );
      currentAffinity <<= 1;

      //
      // We need to run at DISPATCH_LEVEL to
      // properly synchronize access to these
      // power structures
      //
      KeRaiseIrql( DISPATCH_LEVEL, &OldIrql );

      Prcb = KeGetCurrentPrcb();
      PState = &(Prcb->PState);
      PerfStates = PState->PerfStates;
      PerfStatesCount =
            PState->PerfStatesCount;
      For (I = PState->KneeThrottlIndex;
            I < PerfStatesCount;
            I++) {

            If (PerfState[I].MinCapacity >=
                  PopCB.Status.Capacity) {

                  Break;

            }

      }
      PState->ThrottleLimitIdnex = I;
```

```
                                        //
                                        // We can revert back to our previous
                                        // IRQL now
                                        //
                                        KeLowerIrql( OldIrql );

                                    }

                                    KeRevertToUserAffinityThread();

                                    <...>

                                <...>

                                }


                            <...>

                            }

                        <...>

                        }
                }
```